目录

[实验一 基本操作实验 12](#_Toc481700026)

[1.1 实验目的 12](#_Toc481700027)

[1.2 实验要求 12](#_Toc481700028)

[1.3 实验原理 12](#_Toc481700029)

[1.3.1 大数据实验一体机 12](#_Toc481700030)

[1.3.2 Linux基本命令 15](#_Toc481700031)

[1.3.3 Vi编辑器 17](#_Toc481700032)

[1.3.4 SSH免密认证 17](#_Toc481700033)

[1.3.5 Java基本命令 18](#_Toc481700034)

[1.3.6 Eclipse集成开发环境 19](#_Toc481700035)

[1.4 实验步骤 19](#_Toc481700036)

[1.4.1 搭建集群服务器 19](#_Toc481700037)

[1.4.2 使用SSH工具登录每台服务器 19](#_Toc481700038)

[1.4.3 添加域名映射 20](#_Toc481700039)

[1.4.4 配置SSH免密登录 21](#_Toc481700040)

[1.4.5 在client服务器开发Java Helloworld程序 23](#_Toc481700041)

[1.4.6 使用Eclipse开发Java Helloworld程序 24](#_Toc481700042)

[实验二 HDFS实验：部署HDFS 26](#_Toc481700043)

[2.1 实验目的 26](#_Toc481700044)

[2.2 实验要求 26](#_Toc481700045)

[2.3 实验原理 27](#_Toc481700046)

[2.3.1 分布式文件系统 27](#_Toc481700047)

[2.3.2 HDFS 27](#_Toc481700048)

[2.3.3 HDFS基本命令 27](#_Toc481700049)

[2.3.4 HDFS适用场景 27](#_Toc481700050)

[2.4 实验步骤 28](#_Toc481700051)

[2.4.1 在master服务器上确定存在hadoop安装目录 28](#_Toc481700052)

[2.4.2 确认集群服务器之间可SSH免密登录 28](#_Toc481700053)

[2.4.3 修改HDFS配置文件 28](#_Toc481700054)

[2.4.4 启动HDFS 29](#_Toc481700055)

[2.4.5 通过查看进程的方式验证HDFS启动成功 29](#_Toc481700056)

[2.4.6 使用client上传文件 29](#_Toc481700057)

[实验三 HDFS实验：读写HDFS文件 30](#_Toc481700058)

[3.1 实验目的 30](#_Toc481700059)

[3.2 实验要求 30](#_Toc481700060)

[3.3 实验原理 30](#_Toc481700061)

[3.3.1 Java Classpath 30](#_Toc481700062)

[3.3.2 Eclipse Hadoop插件 31](#_Toc481700063)

[3.4 实验步骤 32](#_Toc481700064)

[3.4.1 配置client服务器classpath 32](#_Toc481700065)

[3.4.2 在client服务器编写HDFS写程序 32](#_Toc481700066)

[3.4.3 编译并打包HDFS写程序 33](#_Toc481700067)

[3.4.4 执行HDFS写程序 33](#_Toc481700068)

[3.4.5 在client服务器编写HDFS读程序 33](#_Toc481700069)

[3.4.6 编译并打包HDFS读程序 33](#_Toc481700070)

[3.4.7 执行HDFS读程序 34](#_Toc481700071)

[3.4.8 安装与配置Eclipse Hadoop插件 34](#_Toc481700072)

[3.4.9 使用Eclipse开发并打包HDFS写文件程序 36](#_Toc481700073)

[3.4.10 上传HDFS写文件程序jar包并执行 37](#_Toc481700074)

[3.4.11 使用Eclipse开发并打包HDFS读文件程序 38](#_Toc481700075)

[3.4.12 上传HDFS读文件程序jar包并执行 38](#_Toc481700076)

[实验四 YARN实验：部署YARN集群 39](#_Toc481700077)

[4.1 实验目的 39](#_Toc481700078)

[4.2 实验要求 39](#_Toc481700079)

[4.3 实验原理 39](#_Toc481700080)

[4.3.1 YARN概述 39](#_Toc481700081)

[4.3.2 YARN运行流程 40](#_Toc481700082)

[4.4 实验步骤 40](#_Toc481700083)

[4.4.1 在master机上配置YARN 40](#_Toc481700084)

[4.4.2 统一启动YARN 41](#_Toc481700085)

[4.4.3 验证YARN启动成功 41](#_Toc481700086)

[4.4.4 在client机上提交DistributedShell任务 41](#_Toc481700087)

[4.4.5 在client机上提交MapReduce型任务 42](#_Toc481700088)

[4.5 实验结果 42](#_Toc481700089)

[实验五 MapReduce实验：单词计数 44](#_Toc481700090)

[5.1 实验目的 44](#_Toc481700091)

[5.2 实验要求 44](#_Toc481700092)

[5.3 实验原理 44](#_Toc481700093)

[5.3.1 MapReduce编程 45](#_Toc481700094)

[5.3.2 Java API解析 45](#_Toc481700095)

[5.4 实验步骤 45](#_Toc481700096)

[5.4.1 启动Hadoop 45](#_Toc481700097)

[5.4.2 验证HDFS上没有wordcount的文件夹 45](#_Toc481700098)

[5.4.3 上传数据文件到HDFS 46](#_Toc481700099)

[5.4.4 编写MapReduce程序 46](#_Toc481700100)

[5.4.5 使用Eclipse开发工具将该代码打包 47](#_Toc481700101)

[5.5 实验结果 47](#_Toc481700102)

[5.5.1 程序运行成功控制台上的显示内容 47](#_Toc481700103)

[5.5.2 在HDFS上查看结果 48](#_Toc481700104)

[实验六 MapReduce实验：二次排序 48](#_Toc481700105)

[6.1 实验目的 48](#_Toc481700106)

[6.2 实验要求 49](#_Toc481700107)

[6.3 实验原理 49](#_Toc481700108)

[6.4 实验步骤 49](#_Toc481700109)

[6.4.1 编写程序 49](#_Toc481700110)

[6.4.2 打包提交 53](#_Toc481700111)

[6.5 实验结果 53](#_Toc481700112)

[6.5.1 输入数据 53](#_Toc481700113)

[6.5.2 执行结果 53](#_Toc481700114)

[实验七 MapReduce实验：计数器 54](#_Toc481700115)

[7.1 实验目的 54](#_Toc481700116)

[7.2 实验要求 54](#_Toc481700117)

[7.3 实验背景 54](#_Toc481700118)

[7.3.1 MapReduce计数器是什么？ 54](#_Toc481700119)

[7.3.2 MapReduce计数器能做什么？ 54](#_Toc481700120)

[7.3.3 内置计数器 55](#_Toc481700121)

[7.3.4 计数器使用 55](#_Toc481700122)

[7.3.5 自定义计数器 56](#_Toc481700123)

[7.4 实验步骤 56](#_Toc481700124)

[7.4.1 实验分析设计 56](#_Toc481700125)

[7.4.2 编写程序 56](#_Toc481700126)

[7.4.3 打包并提交 58](#_Toc481700127)

[7.5 实验结果 58](#_Toc481700128)

[7.5.1 输入数据 58](#_Toc481700129)

[7.5.2 输出显示 58](#_Toc481700130)

[实验八 MapReduce实验：Join操作 59](#_Toc481700131)

[8.1 实验目的 59](#_Toc481700132)

[8.2 实验要求 59](#_Toc481700133)

[8.3 实验背景 59](#_Toc481700134)

[8.3.1 概述 59](#_Toc481700135)

[8.3.2 原理 59](#_Toc481700136)

[8.3 实验步骤 60](#_Toc481700137)

[8.3.1 准备阶段 60](#_Toc481700138)

[8.3.2 编写程序 60](#_Toc481700139)

[8.3.3 打包并提交 64](#_Toc481700140)

[8.4 实验结果 64](#_Toc481700141)

[8.4.1 输入数据 64](#_Toc481700142)

[8.4.2 输出显示 65](#_Toc481700143)

[实验九 MapReduce实验：分布式缓存 65](#_Toc481700144)

[9.1 实验目的 65](#_Toc481700145)

[9.2 实验要求 65](#_Toc481700146)

[9.3 实验步骤 66](#_Toc481700147)

[9.3.1 准备数据 66](#_Toc481700148)

[9.3.2 上传数据 66](#_Toc481700149)

[9.3.3 编写代码 67](#_Toc481700150)

[9.3.4 执行代码 70](#_Toc481700151)

[9.3.5 查看结果 70](#_Toc481700152)

[9.4 实验结果 70](#_Toc481700153)

[实验十 Hive实验：部署Hive 71](#_Toc481700154)

[10.1 实验目的 71](#_Toc481700155)

[10.2 实验要求 71](#_Toc481700156)

[10.3 实验原理 71](#_Toc481700157)

[10.4 实验步骤 72](#_Toc481700158)

[10.4.1 安装部署 72](#_Toc481700159)

[10.4.2 配置HDFS 72](#_Toc481700160)

[10.4.3 启动Hive 73](#_Toc481700161)

[10.5 实验结果 73](#_Toc481700162)

[10.5.1 启动结果 73](#_Toc481700163)

[10.5.2 Hive基本命令 73](#_Toc481700164)

[实验十一 Hive实验：新建Hive表 74](#_Toc481700165)

[11.1 实验目的 74](#_Toc481700166)

[11.2 实验要求 74](#_Toc481700167)

[11.3 实验原理 74](#_Toc481700168)

[11.4 实验步骤 75](#_Toc481700169)

[11.4.1 启动Hive 75](#_Toc481700170)

[11.4.2 创建表 75](#_Toc481700171)

[11.4.3 显示表 75](#_Toc481700172)

[11.4.4 显示表列 75](#_Toc481700173)

[11.4.5 更改表 75](#_Toc481700174)

[11.4.6 删除表（或列） 75](#_Toc481700175)

[11.5 实验结果 76](#_Toc481700176)

[实验十二 Hive实验：Hive分区 77](#_Toc481700177)

[12.1 实验目的 77](#_Toc481700178)

[12.2 实验要求 77](#_Toc481700179)

[12.3 实验原理 78](#_Toc481700180)

[12.4 实验步骤 78](#_Toc481700181)

[12.4.1 启动Hadoop集群 78](#_Toc481700182)

[12.4.2 用命令进入Hive客户端 78](#_Toc481700183)

[12.4.3 通过HQL语句进行实验 78](#_Toc481700184)

[12.5 实验结果 79](#_Toc481700185)

[实验十三 Spark实验：部署Spark集群 80](#_Toc481700186)

[13.1 实验目的 80](#_Toc481700187)

[13.2 实验要求 80](#_Toc481700188)

[13.3 实验原理 81](#_Toc481700189)

[13.3.1 Spark简介 81](#_Toc481700190)

[13.3.2 Spark适用场景 81](#_Toc481700191)

[13.4 实验内容和步骤 81](#_Toc481700192)

[13.4.1 配置Spark集群 81](#_Toc481700193)

[13.4.2 配置HDFS 82](#_Toc481700194)

[13.4.3 提交Spark任务 82](#_Toc481700195)

[13.5 实验结果 83](#_Toc481700196)

[13.5.1 进程查看 83](#_Toc481700197)

[13.5.2 验证WebUI 83](#_Toc481700198)

[12.5.3 SparkWordcount程序执行 83](#_Toc481700199)

[实验十四 Spark实验：SparkWordCount 84](#_Toc481700200)

[14.1 实验目的 84](#_Toc481700201)

[14.2 实验要求 84](#_Toc481700202)

[14.3 实验原理 84](#_Toc481700203)

[14.3.1 Scala是兼容的 85](#_Toc481700204)

[14.3.2 Scala是简洁的 85](#_Toc481700205)

[14.3.3 Scala是高级的 86](#_Toc481700206)

[14.3.4 Scala是静态类型的 87](#_Toc481700207)

[14.4 实验步骤 88](#_Toc481700208)

[14.5 实验结果 88](#_Toc481700209)

[实验十五 Spark实验：RDD综合实验 89](#_Toc481700210)

[15.1 实验目的 89](#_Toc481700211)

[15.2 实验要求 89](#_Toc481700212)

[15.3 实验原理 89](#_Toc481700213)

[15.4 实验步骤 90](#_Toc481700214)

[15.4.1 distinct 去除RDD内的重复数据 90](#_Toc481700215)

[15.4.2 foreach 遍历RDD内的数据 90](#_Toc481700216)

[15.4.3 first 取的RDD中的第一个数据 91](#_Toc481700217)

[15.4.4 max 取得RDD中的最大的数据 91](#_Toc481700218)

[15.4.5 intersection 返回两个RDD重叠的数据 91](#_Toc481700219)

[实验十六 Spark实验：Spark综例 92](#_Toc481700220)

[16.1 实验目的 92](#_Toc481700221)

[16.2 实验要求 92](#_Toc481700222)

[16.3 实验原理 92](#_Toc481700223)

[16.3.1 Scala 92](#_Toc481700224)

[16.3.2 Spark Shell 93](#_Toc481700225)

[16.4 实验步骤 93](#_Toc481700226)

[16.4.1 启动Spark Shell 93](#_Toc481700227)

[16.4.2 编写并执行Scala代码 94](#_Toc481700228)

[16.4.3 退出Spark Shell 95](#_Toc481700229)

[16.4.4 查看执行结果 95](#_Toc481700230)

[实验十七 Spark实验：Spark SQL 96](#_Toc481700231)

[17.1 实验目的 96](#_Toc481700232)

[17.2 实验要求 96](#_Toc481700233)

[17.3 实验原理 96](#_Toc481700234)

[17.4 实验步骤 97](#_Toc481700235)

[17.5 实验结果 98](#_Toc481700236)

[实验十八 Spark实验：Spark Streaming 99](#_Toc481700237)

[18.1 实验目的 99](#_Toc481700238)

[18.2 实验要求 99](#_Toc481700239)

[18.3 实验原理 99](#_Toc481700240)

[18.3.1 Spark Streaming架构 99](#_Toc481700241)

[18.3.2 Spark Streaming编程模型 101](#_Toc481700242)

[18.3.3 Spark Streaming典型案例 102](#_Toc481700243)

[18.4 实验步骤 103](#_Toc481700244)

[18.5 实验结果 106](#_Toc481700245)

[实验十九 Spark实验：GraphX 106](#_Toc481700246)

[19.1 实验目的 106](#_Toc481700247)

[19.2 实验要求 106](#_Toc481700248)

[19.3 实验原理 107](#_Toc481700249)

[19.4 实验步骤 107](#_Toc481700250)

[19.4.1 在Intellij IDEA 中安装Scala的插件 107](#_Toc481700251)

[19.4.2 新建Scala Module 107](#_Toc481700252)

[19.4.3 添加maven依赖 107](#_Toc481700253)

[19.4.4 新建Scala程序 107](#_Toc481700254)

[19.4.5 程序运行 111](#_Toc481700255)

[19.5 实验结果 111](#_Toc481700256)

[实验二十 部署ZooKeeper 112](#_Toc481700257)

[20.1 实验目的 112](#_Toc481700258)

[20.2 实验要求 112](#_Toc481700259)

[20.3 实验原理 112](#_Toc481700260)

[20.4 实验步骤 112](#_Toc481700261)

[20.4.1 安装JDK 113](#_Toc481700262)

[20.4.2 修改ZooKeeper配置文件 113](#_Toc481700263)

[20.4.3 启动ZooKeeper集群 113](#_Toc481700264)

[20.5 实验结果 114](#_Toc481700265)

[实验二十一 ZooKeeper进程协作 115](#_Toc481700266)

[21.1 实验目的 115](#_Toc481700267)

[21.2 实验要求 115](#_Toc481700268)

[21.3 实验原理 116](#_Toc481700269)

[21.4 实验步骤 116](#_Toc481700270)

[21.4.1 启动ZooKeeper集群 116](#_Toc481700271)

[21.4.2 导入jar包 116](#_Toc481700272)

[21.4.3 编写java代码 116](#_Toc481700273)

[21.4.4 做成jar包 117](#_Toc481700274)

[21.5 实验结果 117](#_Toc481700275)

[实验二十二 部署HBase 118](#_Toc481700276)

[22.1 实验目的 118](#_Toc481700277)

[22.2 实验要求 118](#_Toc481700278)

[22.3 实验原理 118](#_Toc481700279)

[22.4 实验步骤 119](#_Toc481700280)

[22.5 实验结果 120](#_Toc481700281)

[实验二十三 新建HBase表 121](#_Toc481700282)

[23.1 实验目的 121](#_Toc481700283)

[23.2 实验要求 122](#_Toc481700284)

[23.3 实验原理 122](#_Toc481700285)

[23.4 实验步骤 122](#_Toc481700286)

[23.5 实验结果 127](#_Toc481700287)

[实验二十四 部署Storm 128](#_Toc481700288)

[24.1 实验目的 128](#_Toc481700289)

[24.2 实验要求 128](#_Toc481700290)

[24.3 实验原理 128](#_Toc481700291)

[24.4 实验步骤 129](#_Toc481700292)

[24.5 实验结果 130](#_Toc481700293)

[实验二十五 实时WordCountTopology 131](#_Toc481700294)

[25.1 实验目的 131](#_Toc481700295)

[25.2 实验要求 131](#_Toc481700296)

[25.3 实验原理 132](#_Toc481700297)

[25.3.1 Topologies 132](#_Toc481700298)

[25.3.2 Spouts 132](#_Toc481700299)

[25.3.3 Bolts 133](#_Toc481700300)

[25.4 实验步骤 133](#_Toc481700301)

[25.5 实验结果 136](#_Toc481700302)

[实验二十六 文件数据Flume至HDFS 137](#_Toc481700303)

[26.1 实验目的 137](#_Toc481700304)

[26.2 实验要求 137](#_Toc481700305)

[26.3 实验原理 137](#_Toc481700306)

[26.3.1 flume的特点 138](#_Toc481700307)

[26.3.2 flume的可靠性 138](#_Toc481700308)

[26.4 实验步骤 139](#_Toc481700309)

[26.5 实验结果 140](#_Toc481700310)

[实验二十七 Kafka订阅推送示例 141](#_Toc481700311)

[27.1 实验目的 141](#_Toc481700312)

[27.2 实验要求 141](#_Toc481700313)

[27.3 实验原理 141](#_Toc481700314)

[27.3.1 Kafka简介 141](#_Toc481700315)

[27.3.2 Kafka使用场景 142](#_Toc481700316)

[27.4 实验步骤 143](#_Toc481700317)

[27.4.1 安装Zookeeper集群 143](#_Toc481700318)

[27.4.2 安装Kafka集群 143](#_Toc481700319)

[27.4.3 验证消息推送 144](#_Toc481700320)

[27.5 实验结果 144](#_Toc481700321)

[实验二十八 Pig版WordCount 145](#_Toc481700322)

[28.1 实验目的 145](#_Toc481700323)

[28.2 实验要求 145](#_Toc481700324)

[28.3 实验原理 145](#_Toc481700325)

[28.4 实验步骤 146](#_Toc481700326)

[28.5 实验结果 148](#_Toc481700327)

[实验二十九 Redis部署与简单使用 150](#_Toc481700328)

[29.1 实验目的 150](#_Toc481700329)

[29.2 实验要求 150](#_Toc481700330)

[29.3 实验原理 150](#_Toc481700331)

[29.3.1 CentOS 简介 150](#_Toc481700332)

[29.3.2 CentOS与RHEL关系 150](#_Toc481700333)

[29.3.3 make简介 151](#_Toc481700334)

[29.3.4 Redis简介 151](#_Toc481700335)

[29.4 实验步骤 152](#_Toc481700336)

[29.4.1 安装配置启动 152](#_Toc481700337)

[29.4.2 使用Redis 152](#_Toc481700338)

[实验三十 MapReduce与Spark读写Redis 153](#_Toc481700339)

[30.1 实验目的 153](#_Toc481700340)

[30.2 实验要求 153](#_Toc481700341)

[30.3 实验原理 153](#_Toc481700342)

[30.4 实验步骤 154](#_Toc481700343)

[30.4.1 MapReduce读取Redis 154](#_Toc481700344)

[30.4.2 Spark读取Redis 158](#_Toc481700345)

[30.5 实验结果 160](#_Toc481700346)

[30.5.1 MapReduce读取Redis实验 160](#_Toc481700347)

[30.5.2 Spark读取Redis实验 161](#_Toc481700348)

[实验三十一 MongoDB实验：读写MongoDB 161](#_Toc481700349)

[31.1 实验目的 161](#_Toc481700350)

[31.2 实验要求 161](#_Toc481700351)

[31.3 实验原理 161](#_Toc481700352)

[31.4 实验步骤 162](#_Toc481700353)

[31.4.1 启动MongoDB 162](#_Toc481700354)

[31.4.2 连接使用MongoDB 162](#_Toc481700355)

[31.4.3 连接启动MongoDB的shell，执行一些简单命令。 162](#_Toc481700356)

[实验三十二 LevelDB实验：读写LevelDB 166](#_Toc481700357)

[32.1 实验目的 166](#_Toc481700358)

[32.2 实验要求 166](#_Toc481700359)

[32.3 实验原理 166](#_Toc481700360)

[32.4 实验步骤 168](#_Toc481700361)

[32.4.1 使用C++代码建立数据库连接 169](#_Toc481700362)

[32.4.2 写入数据 169](#_Toc481700363)

[32.4.3 读取数据 169](#_Toc481700364)

[32.4.4 删除数据 169](#_Toc481700365)

[32.4.5 关闭连接 169](#_Toc481700366)

[32.4.6 完整的代码 169](#_Toc481700367)

[32.5 实验结果 170](#_Toc481700368)

[实验三十三 Mahout实验：K-means 171](#_Toc481700369)

[33.1 实验目的 171](#_Toc481700370)

[33.2 实验要求 171](#_Toc481700371)

[33.3 实验原理 171](#_Toc481700372)

[33.3.1 Mahout简介 171](#_Toc481700373)

[33.3.2 Mahout发展 171](#_Toc481700374)

[33.3.3 Mahout特性 172](#_Toc481700375)

[33.3.4 K-means算法概要 172](#_Toc481700376)

[33.3.5 K-means算法存在的问题 173](#_Toc481700377)

[33.3.6 K-means算法优点 173](#_Toc481700378)

[33.3.7 K-means算法缺点 173](#_Toc481700379)

[33.3.8 K-means算法应用 174](#_Toc481700380)

[33.4 实验步骤 174](#_Toc481700381)

[33.4.1 添加临时JAVA\_HOME环境变量 174](#_Toc481700382)

[33.4.2 建立HDFS目录 174](#_Toc481700383)

[33.4.3 实验数据准备 174](#_Toc481700384)

[33.4.4 提交Mahout的K-means程序 174](#_Toc481700385)

[33.5 实验结果 174](#_Toc481700386)

[实验三十四 使用Spark实现K-Means 175](#_Toc481700387)

[34.1 实验目的 175](#_Toc481700388)

[34.2 实验要求 175](#_Toc481700389)

[34.3 实验原理 175](#_Toc481700390)

[34.4 实验步骤 175](#_Toc481700391)

[34.4.1 添加临时JAVA\_HOME环境变量 175](#_Toc481700392)

[34.3.2 上传训练数据集 176](#_Toc481700393)

[34.3.3 训练SVM模型 176](#_Toc481700394)

[34.5 实验结果 177](#_Toc481700395)

[实验三十五 使用Spark实现SVM 178](#_Toc481700396)

[35.1 实验目的 178](#_Toc481700397)

[35.2 实验要求 178](#_Toc481700398)

[35.3 实验原理 178](#_Toc481700399)

[35.3.1 SVM算法介绍 178](#_Toc481700400)

[35.3.1 SVM算法原理 178](#_Toc481700401)

[35.4 实验步骤 179](#_Toc481700402)

[35.3.1 添加临时JAVA\_HOME环境变量 180](#_Toc481700403)

[35.3.2 上传训练数据集 180](#_Toc481700404)

[35.3.3 训练SVM模型 180](#_Toc481700405)

[35.5 实验结果 181](#_Toc481700406)

[实验三十六 使用Spark实现FP-Growth 181](#_Toc481700407)

[36.1 实验目的 181](#_Toc481700408)

[36.2 实验要求 182](#_Toc481700409)

[36.3 实验原理 182](#_Toc481700410)

[36.3.1 FP-Growth算法简介 182](#_Toc481700411)

[36.3.2 FP-Growth算法流程 183](#_Toc481700412)

[36.4 实验步骤 183](#_Toc481700413)

[35.3.1 添加临时JAVA\_HOME环境变量 183](#_Toc481700414)

[35.3.2 上传训练数据集 183](#_Toc481700415)

[36.3.2 训练SVM模型 184](#_Toc481700416)

[36.5 实验结果 185](#_Toc481700417)

[实验三十七 综合实战：车牌识别 186](#_Toc481700418)

[37.1 实验目的 186](#_Toc481700419)

[37.2 实验要求 186](#_Toc481700420)

[37.3 实验步骤 186](#_Toc481700421)

[37.3.1 编写程序 186](#_Toc481700422)

[37.3.2 环境准备 191](#_Toc481700423)

[37.3.3 打包提交 192](#_Toc481700424)

[37.4 实验结果 192](#_Toc481700425)

[37.4.1 输入数据 192](#_Toc481700426)

[37.4.2 执行结果 193](#_Toc481700427)

[实验三十八 综合实战：搜索引擎 194](#_Toc481700428)

[38.1 实验目的 194](#_Toc481700429)

[38.2 实验要求 194](#_Toc481700430)

[38.3 实验步骤 194](#_Toc481700431)

[38.3.1 新建Java项目 195](#_Toc481700432)

[38.3.2 新建JavaWeb项目 200](#_Toc481700433)

[38.3.3 网页扒取 217](#_Toc481700434)

[38.3.4 建立关键词索引 218](#_Toc481700435)

[38.3.5 关键词搜索 219](#_Toc481700436)

[38.4 实验结果 219](#_Toc481700437)

[实验三十九 综合实战：推荐系统 221](#_Toc481700438)

[39.1 实验目的 221](#_Toc481700439)

[39.2 实验要求 221](#_Toc481700440)

[39.3 实验步骤 221](#_Toc481700441)

[39.3.1 试验原理概述 221](#_Toc481700442)

[39.3.2 数据集准备 222](#_Toc481700443)

[39.3.3 代码实现 224](#_Toc481700444)

[39.4 实验结果 227](#_Toc481700445)

[实验四十 综合实战：环境大数据 228](#_Toc481700446)

[40.1 实验目的 228](#_Toc481700447)

[40.2 实验要求 228](#_Toc481700448)

[40.3 实验原理 229](#_Toc481700449)

[40.4 实验步骤 229](#_Toc481700450)

[40.4.1 分析数据文件 229](#_Toc481700451)

[40.4.2 将数据文件上传至HDFS 229](#_Toc481700452)

[40.4.3 编写月平均气温统计程序 230](#_Toc481700453)

[40.4.4 查看月平均气温统计结果 231](#_Toc481700454)

[40.4.5 编写每日空气质量统计程序 232](#_Toc481700455)

[40.4.6 查看每日空气质量统计结果 234](#_Toc481700456)

[40.4.7 将每日空气质量统计文件进行整合 235](#_Toc481700457)

[40.4.8 编写各空气质量天数统计程序 235](#_Toc481700458)

[40.4.9 查看各空气质量天数统计结果 238](#_Toc481700459)

[实验四十一 综合实战：智能硬件大数据托管 239](#_Toc481700460)

[41.1 实验目的 239](#_Toc481700461)

[41.2 实验要求 239](#_Toc481700462)

[41.3 实验原理 239](#_Toc481700463)

[41.3 实验步骤 241](#_Toc481700464)

[41.3.1 万物云平台相关注册 241](#_Toc481700465)

[41.3.2 建表-用于存储智能硬件的数据 242](#_Toc481700466)

[41.3.3 智能硬件接入平台 242](#_Toc481700467)

[41.3.4 数据上传 244](#_Toc481700468)

[41.3.5 数据查询 244](#_Toc481700469)

[41.3.6 简单的数据分析 245](#_Toc481700470)

[41.3 实验结果 246](#_Toc481700471)

[实验四十二 综合实战：贷款风险评估 247](#_Toc481700472)

[42.1 实验目的 247](#_Toc481700473)

[42.2 实验要求 247](#_Toc481700474)

[42.3 实验原理 247](#_Toc481700475)

[42.3.1 分类过程及评估指标 247](#_Toc481700476)

[42.3.2 spark-submit使用详解 248](#_Toc481700477)

[42.4 实验相关 249](#_Toc481700478)

[42.4.1 实验环境 249](#_Toc481700479)

[42.4.2 实验数据 249](#_Toc481700480)

[42.4.3 实验步骤 249](#_Toc481700481)

[42.5 实验结果 254](#_Toc481700482)

实验一 基本操作实验

1.1 实验目的

1． 熟悉大数据实验一体机并了解如何搭建集群；

2． 熟悉Linux基本命令；

3． 掌握vi编辑器的使用；

4． 了解SSH免密登录的原理以及为何需要配置SSH免密登录；

5． 掌握如何配置SSH免密登录；

6． 掌握Java基本命令；

7． 熟悉集成开发软件Eclipse的安装和使用。

1.2 实验要求

本次实验完成后，要求学生能够：

1． 使用大数据实验一体机搭建自己的集群；

2． 通过SSH工具登录集群服务器；

3． 实现每台服务器相互之间的免密登录；

4． 通过vi编辑器编写Java程序；

5． 通过Java命令编译和运行编写的Java程序；

6． 通过Jar命令打包编写的Java程序；

7． 安装Eclipse并在其中编写Java程序。

1.3 实验原理

1.3.1 大数据实验环境

随着移动互联网、云计算、物联网的快速发展，特别是智能手机端博客、社交网络、位置服务（LBS）等信息发布方式的不断涌现，数据正以前所未有的速度在不断地增长和累积，大数据时代已经来到。

在海量数据面前，大数据人才无疑是其中最关键环节之一，然而，不论国内外，大数据人才却紧缺相当稀缺，比如，当前我国数据人才缺口高达150万，而在未来5-10年，随着市场规模不断增加，这一缺口还将不断增加。

然而，在创新探索大数据教学面前，高校却碰到了一系列困难，如大部分高校大数据课程体系并不完善，在实验环节，由于缺乏实验设备，大数据实训案例匮乏、实验难开展。

针对大数据专业建设的三大难题，云创大数据为各大高校量身定制了大数据软硬件一体化的教学科研平台——大数据实验一体机。大数据实验一体机通过应用容器技术，以少量机器虚拟大量实验集群，可供大量学生同时拥有多套集群进行试验，而每个学生的实验环境不仅相互隔离，方便高效地完成实验，而且实验彼此不干扰，即使某个实验环境被破坏，对其他人也没有影响，一键重启就可以拥有一套新集群，大幅度节省了硬件和人员管理的投入成本

此外，作为一个可供大量学生完成大数据与云计算实验的集成环境，该平台同步提供了配套的培训服务，对于教学组件的安装、配置，教材、实验手册等具体应用提供一站式服务，有助于高校更好地满足课程设计、课程上机实验、实习实训、科研训练等多方面需求，并在一定程度上缓解大数据师资不足的问题。对于各大高校而言，即使没有任何大数据实验基础，该平台也能助其轻松开展大数据与云计算的教学、实验与科研。

具体而言，大数据实验一体机从以下三个方面解决了高校大数据的教学科研难题：

（1）完整的大数据课程体系及配套资源，一步解决入门难的问题

在《实战Hadoop2.0——从云计算到大数据》和实验手册的指导之下，大数据实验一体机解决方案涵盖大数据算法、接口、工具、平台等多方面内容，从大数据监测与收集、大数据存储与处理、大数据分析与挖掘直至大数据创新，帮助高校构建完善的大数据课程体系。

综合36个大数据实验的实验手册及配套高清视频课程，涵盖原理验证、综合应用、自主设计及创新的多层次实验内容。每个实验呈现详细的实验目的、实验内容、实验原理和实验流程指导。配套相应的实验数据和高清视频课程，参照手册即可轻松完成每个实验。国内大数据和云计算专业领域排名第一的网站——中国大数据、中国云计算、中国存储等提供全线支持，一网打尽各类优质资源。

（2）安全可靠的实验环境，大幅度提升大数据技能

基于Docker容器技术，大数据实验一体机可瞬间创建随时运行的实验环境。使用几台机器即可虚拟出大量实验集群，方便上百学生同时使用。采用Kubernetes+ZooKeeper架构管理集群，实验集群完全隔离。实验环境互不干扰，如果实验环境破坏，一键重启即可建立新集群。内置数据挖掘等教学实验数据，可导入高校各学科数据进行教学、科研，校外培训机构同样适用。

（3）热门实战项目贯穿始终，进一步提高教学效果与就业率

大数据实验一体机解决方案采用理论与实践相结合的人才培养模式，帮助教师提高教学水平，促使学生完善大数据知识体系。基于真实的企业基地实训经验，提供丰富的项目实训案例。结合高校各专业实际情况进行行业数据研究，培养实用型人才的专业项目能力。

（4）更多潜在效益，同步增强高校的硬实力和影响力

大数据上升为国家战略，发改委明确组建13个国家级大数据实验室，大数据实验一体机有助于高校大数据实验室建设以及高层次大数据人才的深度培育。大数据实验一体机解决方案将理论与实践双管齐下，帮助提升了高校信息化管理水平和实验项目研究水平。大数据产业迎来发展黄金期，以大数据实验一体机提高大数据专业就业率，可进一步增强高校的硬实力和影响力。

在2016年暑期全国高校大数据培训中，云创大数据利用大数据实验一体机搭建了Docker容器云，为每个学员分配5套虚拟服务器集群，为其提供了简洁易用的上机操作环境，得到了学员的一致好评。在理论讲解的基础上，讲师通过这一实践平台，为学员提供精确到每一步的操作指导，真正做到了学思结合、知行统一，所有学员的大数据应用能力均得以提升，并获得了相应的大数据能力等级证书。

大数据实验一体机基本操作主要包括账号管理、集群管理、集群登录和辅助功能四大部分，其中账号管理完成新建和销毁用户账号，集群管理完成新建和销毁集群，集群登录指的是通过SSH登录到集群各机器，辅助功能模板提供了部分软件下载等实用小功能。

1.3.1.1 界面管理

输入本校大数据实验一体机网址后，请输入相应账号与密码，点击登录即可。如图1-1所示：



图1-1 登录界面

1.3.1.2 账号管理

系统管理员和教师角色登录后，可以看到用户账户管理界面。

其中：

系统管理员用户可以在该界面中查看或修改所有的教师和学生用户信息，并可以注册或销毁教师或学生用户账户；

教师用户可以在该界面中查看或修改自己建立的所有学生用户信息，并可以注册或销毁自己的学生用户账户。

如图1-2所示：



图1-2 账号管理

1.3.1.3 集群管理

此处的集群管理包含“创建集群”和“销毁集群”，由于云创大数据实验一体机采用“Docker”技术，因此能够在几乎不占用系统资源情况下，实现大量机器快速创建与销毁，不必担心资源消耗高、启动销毁慢、管理维护难等问题。

（1）创建集群

当需要新建集群时，直接点击集群管理界面的创建集群即可，后台会快速为用户新建五台预安装CentOS 7操作系统的机器，并配置好各自的主机名和IP地址等。如图1-3所示：



图1-3 新建集群

（2）销毁集群

若实验过程中，由于命令敲错等各种原因导致集群无法使用，可在我的主页中随时销毁失效的集群，之后再重新建立新的集群。如图1-4所示：



图1-4 集群主页

1.3.1.4 相关下载

大数据实验一体机的相关下载界面提供了实验所需的软件及插件的下载，为避免软件版本不同导致实验环境配置错误，请尽量下载和使用此处指定的软件版本与插件。如图1-5所示：

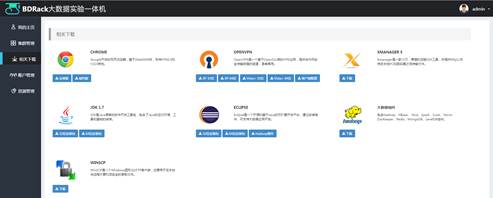


图1-5 相关下载

1.3.2 Linux基本命令

云创大数据实验平台搭建的集群服务器均为预装Linux操作系统的服务器。

Linux是一套免费使用和自由传播的类Unix操作系统，是一个基于POSIX和UNIX的多用户、多任务、支持多线程和多CPU的操作系统。它能运行主要的UNIX工具软件、应用程序和网络协议。它支持32位和64位硬件。Linux继承了Unix以网络为核心的设计思想，是一个性能稳定的多用户网络操作系统。

Linux操作系统诞生于1991年10月5日。Linux存在着许多不同的Linux版本，但它们都使用了Linux内核。Linux可安装在各种计算机硬件设备中，比如手机、平板电脑、路由器、视频游戏控制台、台式计算机、大型机和超级计算机。

严格来讲，Linux这个词本身只表示Linux内核，但实际上人们已经习惯了用Linux来形容整个基于Linux内核，并且使用GNU工程各种工具和数据库的操作系统。

本小节将介绍实验中涉及到的Linux操作系统命令：

(1)查看当前目录

pwd命令用于显示当前目录：

[root@master ~]# pwd

/root

(2)目录切换

cd命令用来切换目录：

[root@master ~]# cd /usr/cstor

[root@master cstor]# pwd

/usr/cstor

[root@master cstor]#

(3)文件罗列

ls命令用于查看文件与目录：

[root@master cstor]# ls

(4)文件或目录拷贝

cp命令用于拷贝文件，若拷贝的对象为目录，则需要使用-r参数：

[root@master cstor]# cp -r hadoop /root/hadoop

(5)文件或目录移动或重命名

mv命令用于移动文件，在实际使用中，也常用于重命名文件或目录：

[root@master ~]# mv hadoop hadoop2 #当前位于/root，不是/usr/cstor

(6)文件或目录删除

rm命令用于删除文件，若删除的对象为目录，则需要使用-r参数：

[root@master ~]# rm -rf hadoop2 #当前位于/root，不是/usr/cstor

(7)进程查看

ps命令用于查看系统的所有进程：

[root@master ~]# ps # 查看当前进程

(8)文件压缩与解压

tar命令用于文件压缩与解压，参数中的c表示压缩，x表示解压缩：

[root@master ~]# tar -zcvf /root/hadoop.tar.gz /usr/cstor/hadoop

[root@master ~]# tar -zxvf /root/hadoop.tar.gz

(9)查看文件内容

cat命令用于查看文件内容：

[root@master ~]# cat /usr/cstor/hadoop/etc/hadoop/core-site.xml

(10)查看服务器IP配置

ip addr命令用于查看服务器IP配置：

[root@master ~]# ip addr

1: lo: <LOOPBACK,UP,LOWER\_UP> mtu 65536 qdisc noqueue state UNKNOWN

link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

inet 127.0.0.1/8 scope host lo

valid\_lft forever preferred\_lft forever

inet6 ::1/128 scope host

valid\_lft forever preferred\_lft forever

125: eth0@if126: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc noqueue state UP

link/ether 02:42:ac:11:00:0c brd ff:ff:ff:ff:ff:ff link-netnsid 0

inet 172.17.0.12/16 scope global eth0

valid\_lft forever preferred\_lft forever

inet6 fe80::42:acff:fe11:c/64 scope link

valid\_lft forever preferred\_lft forever

[root@master ~]#

1.3.3 Vi编辑器

vi编辑器通常被简称为vi，而vi又是visual editor的简称。它在Linux上的地位就像Edit程序在DOS上一样。它可以执行输出、删除、查找、替换、块操作等众多文本操作，而且用户可以根据自己的需要对其进行定制，这是其他编辑程序所没有的。

vi 编辑器并不是一个排版程序，它不像Word或WPS那样可以对字体、格式、段落等其他属性进行编排，它只是一个文本编辑程序。没有菜单，只有命令，且命令繁多。vi有3种基本工作模式：命令行模式、文本输入模式和末行模式。

Vim是vi的加强版，比vi更容易使用。vi的命令几乎全部都可以在vim上使用。

vi编辑器是Linux和Unix上最基本的文本编辑器，工作在字符模式下。由于不需要图形界面，vi是效率很高的文本编辑器。尽管在Linux上也有很多图形界面的编辑器可用，但vi在系统和服务器管理中的功能是那些图形编辑器所无法比拟的。

Vi或vim是实验中用到最多的文件编辑命令，命令行嵌入“vi/vim 文件名”后，默认进入“命令模式”，不可编辑文档，需键盘点击“i”键，方可编辑文档，编辑结束后，需按“ESC”键，先退回命令模式，再按“：”进入末行模式，接着嵌入“wq”方可保存退出。下述的图1-6为vi/vim三种模式转换示意图，图1-7为vi/vim操作实例。

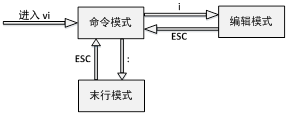


图1-6 vi/vim命令模式

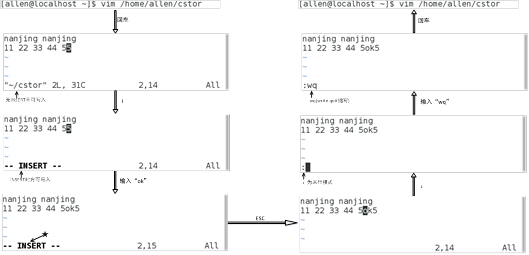


图1-7 vim实例

1.3.4 SSH免密认证

实验中，我们需要从实验室机器登录到集群中的Linux服务器上，而绝大多数Linux服务器采用的是SSH（Secure Shell）登录方式，因此，我们需要在实验室机器上安装一个SSH登录工具。常用的SSH工具包括XShell、Secure CRT、putty等，大数据实验一体机的相关下载界面中提供了XShell工具的下载。

Hadoop的基础是分布式文件系统HDFS，HDFS集群有两类节点以管理者-工作者的模式运行，即一个namenode（管理者）和多个datanode（工作者）。在Hadoop启动以后，namenode通过SSH来启动和停止各个节点上的各种守护进程，这就需要在这些节点之间执行指令时采用无需输入密码的认证方式，因此，我们需要将SSH配置成使用无需输入root密码的密钥文件认证方式。如图1-8所示：

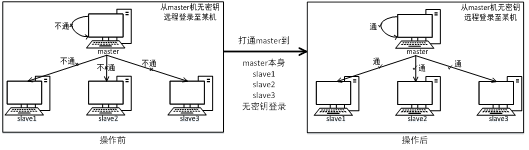


图1-8 实验集群master服务器SSH免密登录示意图

1.3.5 Java基本命令

在安装Java环境后，可以使用Java命令来编译、运行或者打包Java程序。

(1)查看Java版本

[root@client ~]# java -version

java version "1.7.0\_79"

Java(TM) SE Runtime Environment (build 1.7.0\_79-b15)

Java HotSpot(TM) 64-Bit Server VM (build 24.79-b02, mixed mode)

(2)编译Java程序

[root@client ~]# javac Helloworld.java

(3)运行Java程序

[root@client ~]# java Helloworld

Hello World!

(4)打包Java程序

[root@client ~]# jar -cvf Helloworld.jar Helloworld.class

added manifest

adding: Helloworld.class(in = 426) (out= 289)(deflated 32%)

由于打包时并没有指定manifest文件，因此该jar包无法直接运行：

[root@client ~]# java -jar Helloworld.jar

no main manifest attribute, in Helloworld.jar

(5)打包携带manifest文件的Java程序

manifest文件用于描述整个Java项目，最常用的功能是指定项目的入口类：

[root@client ~]# cat manifest.mf

Main-Class: Helloworld

打包时，加入-m参数，并指定manifest文件名：

[root@client ~]# jar -cvfm Helloworld.jar manifest.mf Helloworld.class

added manifest

adding: Helloworld.class(in = 426) (out= 289)(deflated 32%)

之后，即可使用java命令直接运行该jar包：

[root@client ~]# java -jar Helloworld.jar

Hello World!

1.3.6 Eclipse集成开发环境

Eclipse 是一个开放源代码的、基于Java的可扩展开发平台。就其本身而言，它只是一个框架和一组服务，用于通过插件组件构建开发环境。幸运的是，Eclipse 附带了一个标准的插件集，包括Java开发工具（Java Development Kit，JDK）。

Eclipse是著名的跨平台的自由集成开发环境（IDE）。最初主要用来Java语言开发，通过安装不同的插件Eclipse可以支持不同的计算机语言，比如C++和Python等开发工具。Eclipse的本身只是一个框架平台，但是众多插件的支持使得Eclipse拥有其他功能相对固定的IDE软件很难具有的灵活性。许多软件开发商以Eclipse为框架开发自己的IDE。

使用Eclipse可以帮助程序开发人员自动补全语义、方法名、方法参数、语句块等等，并且能够实时检查程序语法，提供错误和警告说明等等，能够极大地提高开发效率。

然而，使用Eclipse会占用较大的系统内存，因此，对于配置不高（32位操作系统或内存不足4G）的实验机器，不推荐安装Eclipse。

1.4 实验步骤

1.4.1 搭建集群服务器

使用自己的账号密码登录大数据实验一体机（默认密码为123456，登录后会自动跳转至密码修改界面，建议修改为自己的密码），进入集群管理界面。如图1-9所示：



图1-9 登录大数据实验一体机

选择第一个Hadoop集群，点击创建集群，等待集群建立完成。如图1-10所示：



图1-10 搭建Hadoop集群完成

1.4.2 使用SSH工具登录每台服务器

在搭建好的Hadoop集群中，已经给出了所有五台服务器的内部ip地址、ssh端口号、ssh登录名以及ssh登录密码。

要想登录这些服务器，我们需要先下载OpenVPN客户端软件。

在相关下载中，根据PC的操作系统版本下载对应版本的OpenVPN客户端安装包，并下载客户端配置文件。如图1-11所示：



图1-11 下载OpenVPN

安装完成后，将下载的客户端配置压缩包解压，将其中的client.ovpn放于OpenVPN安装目录的config文件夹下。

以管理员身份运行OpenVPN GUI，任务栏将出现OpenVPN GUI图标，右键单击任务栏内OpenVPN GUI图标，点击「Connect」。如图1-12所示：

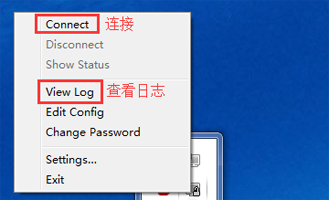


图1-12 登录集群服务器

当提示连接成功后，即可使用SSH工具登录大数据试验一体机分配的内网IP连接你的集群服务器。如图1-13所示：

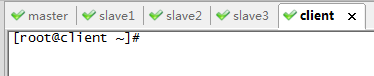


图1-13 登录集群服务器

1.4.3 添加域名映射

系统搭建好的集群服务器已经完成修改主机名、关闭防火墙、安装JDK、同步时钟四步操作，为了可以安装大数据组件，还需为所有机器添加域名映射。如图1-14所示：



图1-14 汇总各机机器名及其对应IP

使用ssh工具登录到master服务器，使用vi命令编辑/etc/hosts文件：

[root@master ~]# vi /etc/hosts #root权限,编辑master的域名映射文件

在文件的末尾追加写入如下五行（具体的IP地址请替换为实际集群服务器内部ip）：

172.17.0.7 master

172.17.0.10 slave1

172.17.0.33 slave2

172.17.0.8 slave3

172.17.0.34 client

保存退出后，master服务器的域名映射即添加完成，使用cat命令查看/etc/hosts文件。如图1-15所示：



图1-15 配置域名映射文件

依次登录slave1~3和client服务器，重复该操作。

1.4.4 配置SSH免密登录

1）生成master服务器密钥

执行命令ssh-keygen，生成master服务器密钥。如图1-16所示：

[root@master ~]# ssh-keygen #root用户,master机,生成公私钥

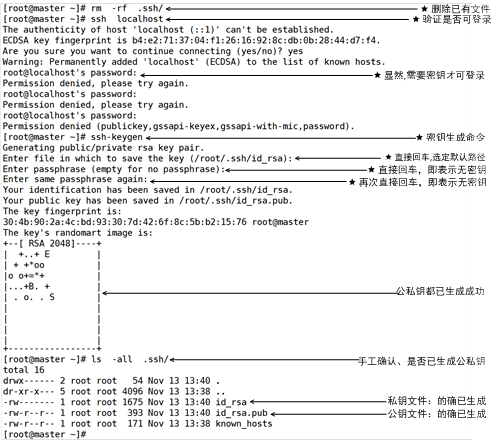


图1-16 master机密钥生成

如图1-14所示，在master上执行“ssh-keygen”命令生成公私钥。第一个提示是询问将公私钥文件存放在哪，直接回车，选择默认位置。

第二个提示是请求用户输入密钥，既然操作的目的就是实现SSH无密钥登录，故此处必须使用空密钥，所谓的空密钥指的是直接回车，不是空格，更不是其他字符。此处请读者务必直接回车，使用空密钥。第三个提示是要求用户确认刚才输入的密钥，既然刚才是空密钥（直接回车即空），那现在也应为空，直接回车即可。

最后，可通过命令“ls -all /root/.ssh”查看到，SSH密钥文件夹.ssh目录下的确生成了两个文件id\_rsa和id\_rsa\_pub，这两个文件都有用，其中公钥用于加密，私钥用于解密。中间的rsa表示算法为RSA算法。

2）拷贝master服务器公钥至本机

执行命令ssh-copy-id master，将master服务器公钥拷贝至master服务器本身。如图1-17所示：

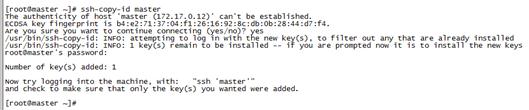


图1-17 拷贝master公钥至master

第一次连接master时，需要输入yes来确认建立授权的主机名访问，并需要输入root用户密码来完成公钥文件传输。

3）验证master服务器ssh免密登录master本身

公钥拷贝完成后，可以在master服务器上直接执行命令ssh master，查看是否可以免密登录master服务器：

[root@master ~]# ssh master #root用户,登录本机网络地址

[root@master ~]# exit #退出本次登录

logout

Connection to master closed.

[root@master ~]#

4）拷贝master服务器公钥至其余服务器

执行命令ssh-copy-id slave1，将master服务器公钥拷贝至slave1服务器。如图1-18所示：

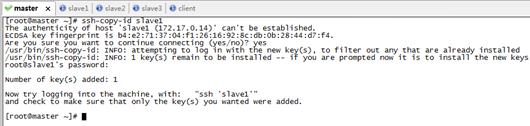


图1-18 拷贝master公钥至slave1

第一次连接slave1时，需要输入yes来确认建立授权的主机名访问，并需要输入root用户密码来完成公钥文件传输。

依照同样的方式将公钥拷贝至slave2、slave3和client服务器。

5）验证master服务器ssh免密登录其余服务器

公钥拷贝完成后，可以在master服务器上直接执行命令ssh master，查看是否可以免密登录slave1~3和client服务器：

[root@master ~]# ssh localhost #root用户,登录本机环回地址

[root@master ~]# ssh master #root用户,登录本机网络地址

[root@master ~]# ssh slave1 #root用户,从master远程登录slave1

[root@master ~]# ssh slave2 #root用户,从master远程登录slave2

[root@master ~]# ssh slave3 #root用户,从master远程登录slave3

6）其余服务器配置ssh免密登录

其余服务器按照同样的方式配置ssh免密登录，完成后验证是否可以互相之间实现SSH免密登录。

1.4.5 在client服务器开发Java Helloworld程序

使用ssh工具登录client服务器，使用vi编辑器编写Helloworld.java：

public class Helloworld {

public static void main(String args[]) {

System.out.println("Hello World!");

}

}

使用javac命令编译该程序，生成Helloworld.class文件：

[root@client ~]# javac Helloworld.java

[root@client ~]# ls

anaconda-ks.cfg data envSource Helloworld.class Helloworld.java

使用java命令执行该程序，输出Hello World!

[root@client ~]# java Helloworld

Hello World!

1.4.6 使用Eclipse开发Java Helloworld程序

根据实验室机器的环境，下载并安装对应版本的JDK和Eclipse软件（若已安装则跳过该步骤）。

安装完成后，双击Eclipse图标，打开该软件。如图1-19所示：



图1-19 Eclipse界面

依次点击FileNewJava Project或FileNewOther Java Project，项目名为Demo。如图1-20所示：

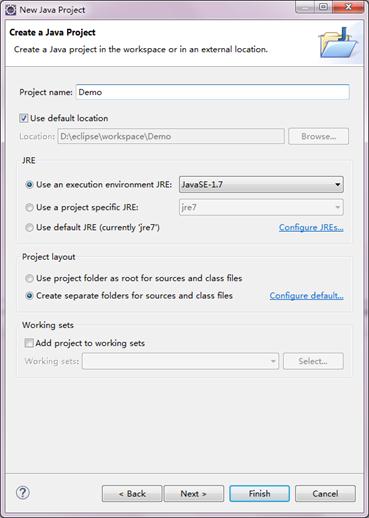


图1-20 创建Java项目

点击Finish，新建项目完成。在左侧导航栏中选中该项目，右键点击，选择New Class，新建Helloworld类，并在该文件入如下内容：

public class Helloworld {

public static void main(String args[]) {

System.out.println("Hello World!");

}

}

完成后，点击上方的Run按钮，即可执行该程序。如图1-21所示：

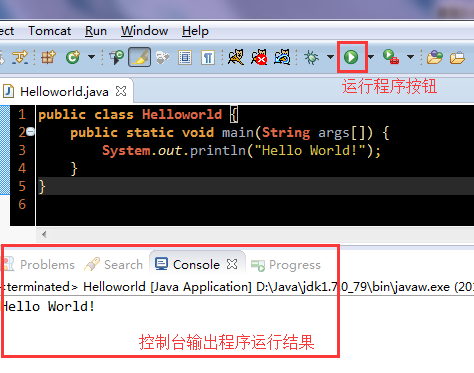


图1-21 Eclipse执行Java程序

实验二 HDFS实验：部署HDFS

2.1 实验目的

1． 理解HDFS存在的原因；

2． 理解HDFS体系架构；

3． 理解master/slave架构；

4． 理解为何配置文件里只需指定主服务、无需指定从服务；

5． 理解为何需要客户端节点；

6． 学会逐一启动HDFS和统一启动HDFS；

7． 学会在HDFS中上传文件。

2.2 实验要求

要求实验结束时，已构建出以下HDFS集群：

1． master上部署主服务NameNode；

2． Slave1、2、3上部署从服务DataNode；

3． client上部署HDFS客户端。

待集群搭建好后，还需在client上进行下述操作：

1． 在HDFS里新建目录；

2． 将client上某文件上传至HDFS里刚才新建的目录。

2.3 实验原理

2.3.1 分布式文件系统

分布式文件系统（Distributed File System）是指文件系统管理的物理存储资源不一定直接连接在本地节点上，而是通过计算机网络与节点相连。该系统架构于网络之上，势必会引入网络编程的复杂性，因此分布式文件系统比普通磁盘文件系统更为复杂。

2.3.2 HDFS

HDFS（Hadoop Distributed File System）为大数据平台其它所有组件提供了最基本的存储功能。它具有高容错、高可靠、可扩展、高吞吐率等特征，为大数据存储和处理提供了强大的底层存储架构。

HDFS是一个主/从（master/slave）体系结构，从最终用户的角度来看，它就像传统的文件系统，可通过目录路径对文件执行CRUD操作。由于其分布式存储的性质，HDFS集群拥有一个NameNode和一些DataNodes，NameNode管理文件系统的元数据，DataNode存储实际的数据。

HDFS开放文件系统的命名空间以便用户以文件形式存储数据，秉承“一次写入、多次读取”的原则。客户端通过NameNode和DataNodes的交互访问文件系统，联系NameNode以获取文件的元数据，而真正的文件I/O操作是直接和DataNode进行交互的。

2.3.3 HDFS基本命令

HDFS基本命令格式如下：

hadoop fs -cmd args

其中，cmd为具体的操作，args为参数。

部分HDFS命令示例如下：

hadoop fs -mkdir /user/trunk #建立目录/user/trunk

hadoop fs -ls /user #查看/user目录下的目录和文件

hadoop fs -lsr /user #递归查看/user目录下的目录和文件

hadoop fs -put test.txt /user/trunk #上传test.txt文件至/user/trunk

hadoop fs -get /user/trunk/test.txt #获取/user/trunk/test.txt文件

hadoop fs -cat /user/trunk/test.txt #查看/user/trunk/test.txt文件内容

hadoop fs -tail /user/trunk/test.txt #查看/user/trunk/test.txt文件的最后1000行

hadoop fs -rm /user/trunk/test.txt #删除/user/trunk/test.txt文件

hadoop fs -help ls #查看ls命令的帮助文档

2.3.4 HDFS适用场景

HDFS 提供高吞吐量应用程序数据访问功能，适合带有大型数据集的应用程序，以下是一些常用的应用场景：

数据密集型并行计算：数据量极大，但是计算相对简单的并行处理，如大规模Web信息搜索；

计算密集型并行计算：数据量相对不是很大，但是计算较为复杂的并行计算，如3D建模与渲染、气象预报和科学计算；

数据密集与计算密集混合型的并行计算，如3D电影的渲染。

HDFS在使用过程中有以下限制：

HDFS不适合大量小文件的存储，因NameNode将文件系统的元数据存放在内存中，因此存储的文件数目受限于NameNode的内存大小；

HDFS适用于高吞吐量，而不适合低时间延迟的访问；

流式读取的方式，不适合多用户写入一个文件（一个文件同时只能被一个客户端写），以及任意位置写入（不支持随机写）；

HDFS更加适合写入一次，读取多次的应用场景。

2.4 实验步骤

部署HDFS主要步骤如下：

1． 配置Hadoop的安装环境；

2． 配置Hadoop的配置文件；

3． 启动HDFS服务；

4． 验证HDFS服务可用。

2.4.1 在master服务器上确定存在hadoop安装目录

[root@master ~]# ls /usr/cstor/hadoop

显示结果如图2-1所示：

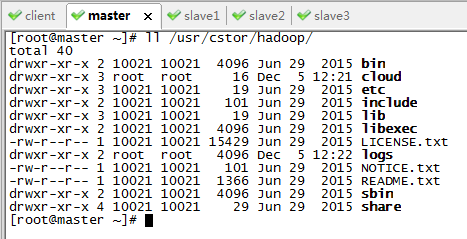


图2-1 确认hadoop安装目录

2.4.2 确认集群服务器之间可SSH免密登录

使用ssh工具登录到每一台服务器，执行命令ssh 主机名，确认每台集群服务器均可SSH免密登录。若无法SSH免密登录，请参照实验一的1.4.4节进行配置。

2.4.3 修改HDFS配置文件

(1)设置JDK安装目录

编辑文件“/usr/cstor/hadoop/etc/hadoop/hadoop-env.sh”，找到如下一行：

export JAVA\_HOME=${JAVA\_HOME}

将这行内容修改为：

export JAVA\_HOME=/usr/local/jdk1.7.0\_79/

这里的“/usr/local/jdk1.7.0\_79/”就是JDK安装位置，如果不同，请根据实际情况更改。

(2)指定HDFS主节点

编辑文件“/usr/cstor/hadoop/etc/hadoop/core-site.xml”，将如下内容嵌入此文件里最后两行的<configuration></configuration>标签之间：

<property><name>hadoop.tmp.dir</name><value>/usr/cstor/hadoop/cloud</value></property>

<property><name>fs.defaultFS</name><value>hdfs://master:8020</value></property>

(3)拷贝集群配置至其它服务器

在master机上执行下列命令，将配置好的hadoop拷贝至slaveX、client。

[root@master ~]# cat ~/data/2/machines

slave1

salve2

slave3

client

[root@master ~]# for x in `cat ~/data/2/machines` ; do echo $x ; scp -r /usr/cstor/hadoop/etc $x:/usr/cstor/hadoop ; done;

2.4.4 启动HDFS

在master服务器上格式化主节点：

[root@master ~]# hdfs namenode -format

配置slaves文件，将localhost修改为slave1~3：

[root@master ~]# vi /usr/cstor/hadoop/etc/hadoop/slaves

slave1

slave2

slave3

统一启动HDFS：

[root@master ~]#cd /usr/cstor/hadoop

[root@master hadoop]# sbin/start-dfs.sh

2.4.5 通过查看进程的方式验证HDFS启动成功

分别在master、slave1~3四台机器上执行如下命令，查看HDFS服务是否已启动。

[root@master sbin]# jps #jps查看java进程

若启动成功，在master上会看到类似的如下信息：

6208 NameNode

6862 Jps

6462 SecondaryNameNode

而在slave1、slave2、slave3上会看到类似的如下信息：

6208 DataNode

6862 Jps

2.4.6 使用client上传文件

从client服务器向HDFS上传文件。

[root@client ~]# hadoop fs -put ~/data/2/machines /

执行命令：hadoop fs -ls /，查看文件是否上传成功。

如图2-2所示：

https://bd.cstor.cn/experiment/exp2.files/image002.jpg

图2-2 查看文件是否上传成功

实验三 HDFS实验：读写HDFS文件

3.1 实验目的

1． 会在Linux环境下编写读写HDFS文件的代码；

2． 会使用jar命令打包代码；

3． 会在client服务器上运行HDFS读写程序；

4． 会在Windows上安装Eclipse Hadoop插件；

5． 会在Eclipse环境编写读写HDFS文件的代码；

6． 会使用Eclipse打包代码；

7． 会使用Xftp工具将实验电脑上的文件上传至client服务器。

3.2 实验要求

要求实验结束时，每位学生均已搭建HDFS开发环境；编写了HDFS写、读代码；在client机上执行了该写、读程序。通过实验了解HDFS读写文件的调用流程，理解HDFS读写文件的原理。

3.3 实验原理

3.3.1 Java Classpath

Classpath设置的目的，在于告诉Java执行环境，在哪些目录下可以找到您所要执行的Java程序所需要的类或者包。

Java执行环境本身就是一个平台，执行于这个平台上的程序是已编译完成的Java程序(后面会介绍到Java程序编译完成之后，会以.class文件存在)。如果将Java执行环境比喻为操作系统，如果设置Path变量是为了让操作系统找到指定的工具程序(以Windows来说就是找到.exe文件)，则设置Classpath的目的就是让Java执行环境找到指定的Java程序(也就是.class文件)。

有几个方法可以设置Classpath，最简单的方法是在系统变量中新增Classpath环境变量。以Windows 7操作系统为例，右键点击计算机 属性 高级系统设置 环境变量，在弹出菜单的“系统变量”下单击“新建”按钮，在“变量名”文本框中输入Classpath，在“变量值”文本框中输入Java类文件的位置。例如可以输入“.; D:\Java\jdk1.7.0\_79\lib\tools.jar; D:\Java\jdk1.7.0\_79\lib\rt.jar”，每一路径中间必须以英文;作为分隔。如图3-1所示：

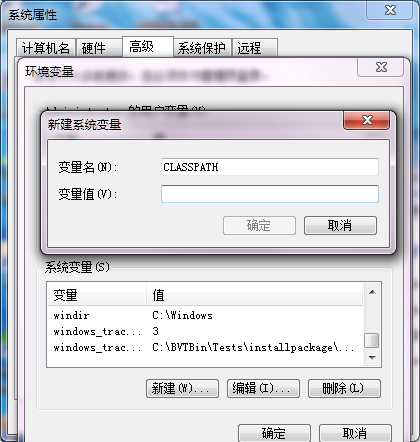


图3-1 Win 7配置Classpath

事实上JDK 7.0默认就会到当前工作目录(上面的.设置)，以及JDK的lib目录(这里假设是D:\Java\jdk1.7.0\_796\lib)中寻找Java程序。所以如果Java程序是在这两个目录中，则不必设置Classpath变量也可以找得到，将来如果Java程序不是放置在这两个目录时，则可以按上述设置Classpath。

如果所使用的JDK工具程序具有Classpath命令选项，则可以在执行工具程序时一并指定Classpath。例如：

javac -classpath classpath1;classpath2...其中classpath1、classpath 2是实际要指定的路径。也可以在命令符模式下执行以下的命令，直接设置环境变量，包括Classpath变量(这个设置在下次重新打开命令符模式时就不再有效)：

set CLASSPATH=%CLASSPATH%;classpath1;classpath2...总而言之，设置Classpath的目的，在于告诉Java执行环境，在哪些目录下可以找到您所要执行的Java程序(.class文件)。

3.3.2 Eclipse Hadoop插件

Eclipse 是一个跨平台的自由集成开发环境（IDE）。通过安装不同的插件，Eclipse可以支持不同的计算机语言，比如C++和Python等开发工具，亦可以通过hadoop插件来扩展开发Hadoop相关程序。

实际工作中，Eclipse Hadoop插件需要根据hadoop集群的版本号进行下载并编译，过程较为繁琐。为了节约时间，将更多的精力用于实现读写HDFS文件，在大数据实验一体机的相关下载页面中已经提供了2.7.1版本的hadoop插件和相关的hadoop包下载，实验人员可以直接下载这些插件，快速在Eclipse中进行安装，开发自己的hadoop程序。

3.4 实验步骤

3.4.1 配置client服务器classpath

使用ssh工具登录client服务器，执行命令vi /etc/profile，编辑该文件，将末尾的如下几行：

JAVA\_HOME=/usr/local/jdk1.7.0\_79/

export JRE\_HOME=/usr/local/jdk1.7.0\_79//jre

export PATH=$PATH:$JAVA\_HOME/bin:$JRE\_HOME/bin

export CLASSPATH=.:$JAVA\_HOME/lib:$JRE\_HOME/lib

export HADOOP\_HOME=/usr/cstor/hadoop

export PATH=$PATH:$HADOOP\_HOME/bin

export HADOOP\_COMMON\_LIB\_NATIVE\_DIR=$HADOOP\_HOME/lib/native

export HADOOP\_OPTS="-Djava.library.path=$HADOOP\_HOME/lib"

用下列行进行替换：

JAVA\_HOME=/usr/local/jdk1.7.0\_79/

export HADOOP\_HOME=/usr/cstor/hadoop

export JRE\_HOME=/usr/local/jdk1.7.0\_79//jre

export PATH=$PATH:$JAVA\_HOME/bin:$JRE\_HOME/bin

export CLASSPATH=.:$JAVA\_HOME/lib:$JRE\_HOME/lib:$HADOOP\_HOME/share/hadoop/common/\*:$HADOOP\_HOME/share/hadoop/common/lib/\*

export PATH=$PATH:$HADOOP\_HOME/bin

export HADOOP\_COMMON\_LIB\_NATIVE\_DIR=$HADOOP\_HOME/lib/native

export HADOOP\_OPTS="-Djava.library.path=$HADOOP\_HOME/lib:$HADOOP\_HOME/lib/native"

执行命令source /etc/profile，使刚才的环境变量修改生效：

[root@client ~]# source /etc/profile

3.4.2 在client服务器编写HDFS写程序

在client服务器上执行命令vi WriteFile.java，编写HDFS写文件程序：

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.FSDataOutputStream;

import org.apache.hadoop.fs.FileSystem;

import org.apache.hadoop.fs.Path;

public class WriteFile {

public static void main(String[] args)throws Exception{

Configuration conf=new Configuration();

FileSystem hdfs = FileSystem.get(conf);

Path dfs = new Path("/weather.txt");

FSDataOutputStream outputStream = hdfs.create(dfs);

outputStream.writeUTF("nj 20161009 23\n");

outputStream.close();

}

}

3.4.3 编译并打包HDFS写程序

使用javac编译刚刚编写的代码，并使用jar命令打包为hdpAction.jar：

[root@client ~]# javac WriteFile.java

[root@client ~]# jar -cvf hdpAction.jar WriteFile.class

added manifest

adding: WriteFile.class(in = 833) (out= 489)(deflated 41%)

3.4.4 执行HDFS写程序

在client服务器上使用hadoop jar命令执行hdpAction.jar：

[root@client ~]# hadoop jar ~/hdpAction.jar WriteFile

查看是否已生成weather.txt文件，若已生成，则查看文件内容是否正确：

[root@client ~]# hadoop fs -ls /

Found 2 items

-rw-r--r-- 3 root supergroup 29 2016-12-05 12:28 /machines

-rw-r--r-- 3 root supergroup 17 2016-12-05 14:54 /weather.txt

[root@client ~]# hadoop fs -cat /weather.txt

nj 20161009 23

3.4.5 在client服务器编写HDFS读程序

在client服务器上执行命令vi ReadFile.java，编写HDFS读文件程序：

import java.io.IOException;

import org.apache.Hadoop.conf.Configuration;

import org.apache.Hadoop.fs.FSDataInputStream;

import org.apache.Hadoop.fs.FileSystem;

import org.apache.Hadoop.fs.Path;

public class ReadFile {

public static void main(String[] args) throws IOException {

Configuration conf = new Configuration();

Path inFile = new Path("/weather.txt");

FileSystem hdfs = FileSystem.get(conf);

FSDataInputStream inputStream = hdfs.open(inFile);

System.out.println("myfile: " + inputStream.readUTF());

inputStream.close();

}

}

3.4.6 编译并打包HDFS读程序

使用javac编译刚刚编写的代码，并使用jar命令打包为hdpAction.jar

[root@client ~]# javac ReadFile.java

[root@client ~]# jar -cvf hdpAction.jar ReadFile.class

added manifest

adding: ReadFile.class(in = 1093) (out= 597)(deflated 45%)

3.4.7 执行HDFS读程序

在client服务器上使用hadoop jar命令执行hdpAction.jar，查看程序运行结果：

[root@client ~]# hadoop jar ~/hdpAction.jar ReadFile

myfile: nj 20161009 23

[root@client ~]#

3.4.8 安装与配置Eclipse Hadoop插件

关闭Eclipse软件，将hadoop-eclipse-plugin-2.7.1.jar文件拷贝至eclipse安装目录的plugins文件夹下。如图3-2和图3-3所示：

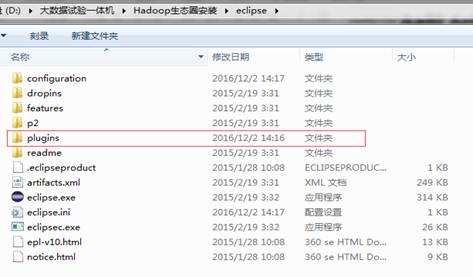


图3-2 Eclipse软件的插件文件夹

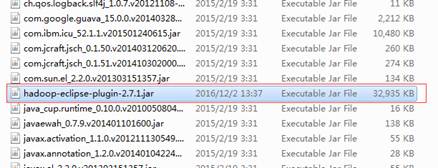


图3-3 将hadoop-eclipse-plugin-2.7.1.jar文件拷贝至插件文件夹中

接下来，我们需要准备本地的Hadoop环境，用于加载hadoop目录中的jar包，只需解压hadoop-2.7.1.tar.gz文件，解压过程中可能会遇到如下错误，点击关闭忽略即可。如图3-4所示：

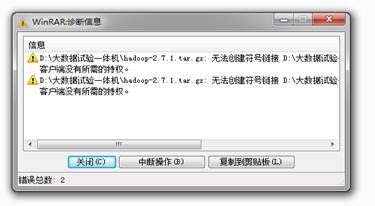


图3-4 解压hadoop-2.7.1.tar.gz可能遇到的错误

现在，我们需要验证是否可以用Eclipse新建Hadoop（HDFS）项目。打开Eclipse软件，依次点击FileNewOther，查看是否已经有Map/Reduce Project的选项。第一次新建Map/Reduce项目时，需要指定hadoop解压后的位置。如图3-5、图3-6和图3-7所示：

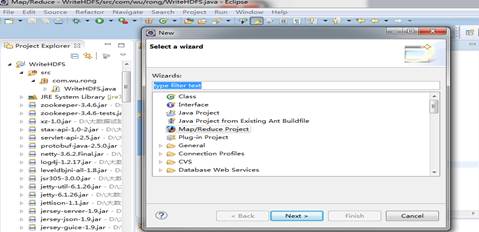


图3-5 Eclipse新建Map/Reduce项目

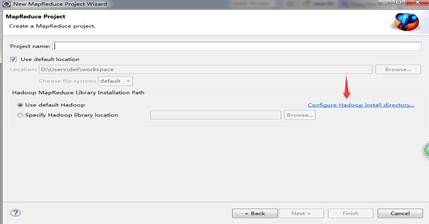


图3-6 设置Hadoop安装目录



图3-7 指定Hadoop安装目录

3.4.9 使用Eclipse开发并打包HDFS写文件程序

打开Eclipse，依次点击FileNewMap/Reduce Project或FileNewOther Map/Reduce Project，新建项目名为WriteHDFS的Map/Reduce项目。

新建WriteFile类并编写如下代码：

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.FSDataOutputStream;

import org.apache.hadoop.fs.FileSystem;

import org.apache.hadoop.fs.Path;

public class WriteFile {

public static void main(String[] args)throws Exception{

Configuration conf=new Configuration();

FileSystem hdfs = FileSystem.get(conf);

Path dfs = new Path("/weather.txt");

FSDataOutputStream outputStream = hdfs.create(dfs);

outputStream.writeUTF("nj 20161009 23\n");

outputStream.close();

}

}

在Eclipse左侧的导航栏选中该项目，点击ExportJavaJAR File，填写导出文件的路径和文件名（本例中设置为hdpAction.jar），确定导出即可。如图3-8和图3-9所示：

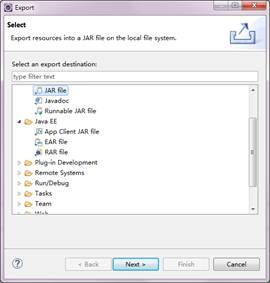


图3-8 选择导出JAR包文件

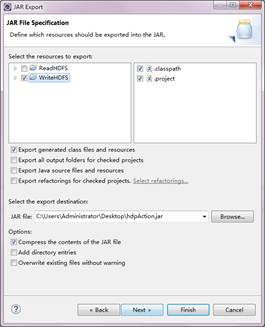


图3-9 指定导出的JAR包文件名

3.4.10 上传HDFS写文件程序jar包并执行

使用WinSCP、XManager或其它SSH工具的sftp工具上传刚刚生成的hdpAction.jar包至client服务器：

sftp> lcd C:/Users/Administrator/Desktop/

sftp> put hdpAction.jar

Uploading hdpAction.jar to /root/hdpAction.jar

100% 2KB 2KB/s 00:00:00

C:/Users/Administrator/Desktop/hdpAction.jar: 2807 bytes transferred in 0 seconds (2807 bytes/s)

在client服务器上使用hadoop jar命令执行hdpAction.jar：

[root@client ~]# hadoop jar ~/hdpAction.jar WriteFile

查看是否已生成weather.txt文件，若已生成，则查看文件内容是否正确：

[root@client ~]# hadoop fs -ls /

Found 2 items

-rw-r--r-- 3 root supergroup 29 2016-12-05 12:28 /machines

-rw-r--r-- 3 root supergroup 17 2016-12-05 14:54 /weather.txt

[root@client ~]# hadoop fs -cat /weather.txt

nj 20161009 23

3.4.11 使用Eclipse开发并打包HDFS读文件程序

打开Eclipse，依次点击FileNewMap/Reduce Project或FileNewOther Map/Reduce Project，新建项目名为ReadHDFS的Map/Reduce项目。

新建ReadFile类并编写如下代码：

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.FSDataInputStream;

import org.apache.hadoop.fs.FileSystem;

import org.apache.hadoop.fs.Path;

public class ReadFile {

public static void main(String[] args) throws IOException {

Configuration conf = new Configuration();

Path inFile = new Path("/weather.txt");

FileSystem hdfs = FileSystem.get(conf);

FSDataInputStream inputStream = hdfs.open(inFile);

System.out.println("myfile: " + inputStream.readUTF());

inputStream.close();

}

}

在Eclipse左侧的导航栏选中该项目，点击ExportJavaJAR File，导出为hdpAction.jar。

3.4.12 上传HDFS读文件程序jar包并执行

使用WinSCP、XManager或其它SSH工具的sftp工具上传刚刚生成的hdpAction.jar包至client服务器，并在client服务器上使用hadoop jar命令执行hdpAction.jar，查看程序运行结果：

[root@client ~]# hadoop jar ~/hdpAction.jar ReadFile

myfile: nj 20161009 23

[root@client ~]#

实验四 YARN实验：部署YARN集群

4.1 实验目的

了解什么是YARN框架，如何搭建YARN分布式集群，并能够使用YARN集群提交一些简单的任务，理解YARN作为Hadoop生态中的资源管理器的意义。

4.2 实验要求

搭建YARN集群，并使用YARN集群提交简单的任务。观察任务提交的之后的YARN的执行过程。

4.3 实验原理

4.3.1 YARN概述

YARN是一个资源管理、任务调度的框架，采用master/slave架构，主要包含三大模块：ResourceManager（RM）、NodeManager（NM）、ApplicationMaster（AM）。其中，ResourceManager负责所有资源的监控、分配和管理，运行在主节点； NodeManager负责每一个节点的维护，运行在从节点；ApplicationMaster负责每一个具体应用程序的调度和协调，只有在有任务正在执行时存在。对于所有的applications，RM拥有绝对的控制权和对资源的分配权。而每个AM则会和RM协商资源，同时和NodeManager通信来执行和监控task。几个模块之间的关系如图4-1所示：

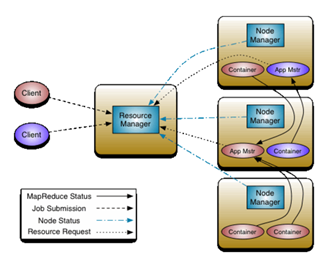


图4-1 模块间的关系

4.3.2 YARN运行流程

YARN运行流程如图4-2所示：

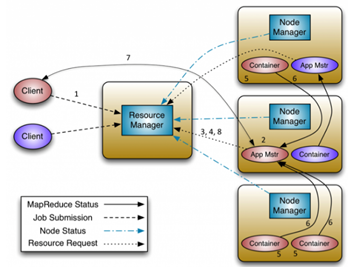


图4-2 YARN运行流程如图

client向RM提交应用程序，其中包括启动该应用的ApplicationMaster的必须信息，例如ApplicationMaster程序、启动ApplicationMaster的命令、用户程序等。

ResourceManager启动一个container用于运行ApplicationMaster。

启动中的ApplicationMaster向ResourceManager注册自己，启动成功后与RM保持心跳。

ApplicationMaster向ResourceManager发送请求，申请相应数目的container。

ResourceManager返回ApplicationMaster的申请的containers信息。申请成功的container，由ApplicationMaster进行初始化。container的启动信息初始化后，AM与对应的NodeManager通信，要求NM启动container。AM与NM保持心跳，从而对NM上运行的任务进行监控和管理。

container运行期间，ApplicationMaster对container进行监控。container通过RPC协议向对应的AM汇报自己的进度和状态等信息。

应用运行期间，client直接与AM通信获取应用的状态、进度更新等信息。

应用运行结束后，ApplicationMaster向ResourceManager注销自己，并允许属于它的container被收回。

4.4 实验步骤

该实验主要分为配置YARN的配置文件，启动YARN集群，向YARN几个简单的任务从而了解YARN工作的流程。

4.4.1 在master机上配置YARN

操作之前请确认HDFS已经启动，具体操作参考之前的实验内容。

指定YARN主节点，编辑文件“/usr/cstor/hadoop/etc/hadoop/yarn-site.xml”，将如下内容嵌入此文件里configuration标签间：

<property><name>yarn.resourcemanager.hostname</name><value>master</value></property>

<property><name>yarn.nodemanager.aux-services</name><value>mapreduce\_shuffle</value></property>

yarn-site.xml是YARN守护进程的配置文件。第一句配置了ResourceManager的主机名，第二句配置了节点管理器运行的附加服务为mapreduce\_shuffle，只有这样才可以运行MapReduce程序。

在master机上操作：将配置好的YARN配置文件拷贝至slaveX、client。

[root@master ~]# cat ~/data/4/machines

slave1

salve2

slave3

client

[allen@cmaster ~]# for x in `cat ~/data/4/machines` ; do echo $x ; scp /usr/cstor/hadoop/etc/hadoop/yarn-site.xml $x:/usr/cstor/hadoop/etc/hadoop/ ; done;

4.4.2 统一启动YARN

确认已配置slaves文件，在master机器上查看：

[root@master ~]# cat /usr/cstor/hadoop/etc/hadoop/slaves

slave1

slave2

slave3

[root@master ~]#

YARN配置无误，统一启动YARN：

[root@master ~]# /usr/cstor/hadoop/sbin/start-yarn.sh

4.4.3 验证YARN启动成功

读者可分别在四台机器上执行如下命令，查看YARN服务是否已启动。

[root@master ~]# jps #jps查看java进程

你会在master上看到类似的如下信息：

2347 ResourceManager

这表明在master节点成功启动ResourceManager，它负责整个集群的资源管理分配，是一个全局的资源管理系统。

而在slave1、slave2、slave3上看到类似的如下信息：

4021 NodeManager

NodeManager是每个节点上的资源和任务管理器，它是管理这台机器的代理，负责该节点程序的运行，以及该节点资源的管理和监控。YARN集群每个节点都运行一个NodeManager。

查看Web界面

在当前的Windows机器上打开浏览器，地址栏输入master的IP和端口号8088（例：10.1.1.7:8088），即可在Web界面看到YARN相关信息。

4.4.4 在client机上提交DistributedShell任务

distributedshell，他可以看做YARN编程中的“hello world”，它的主要功能是并行执行用户提供的shell命令或者shell脚本。-jar指定了包含ApplicationMaster的jar文件，-shell\_command指定了需要被ApplicationMaster执行的Shell命令。

在xshell上再打开一个client 的连接，执行：

[root@client ~]# /usr/cstor/hadoop/bin/yarn

org.apache.hadoop.yarn.applications.distributedshell.Client -jar

/usr/cstor/hadoop/share/hadoop/yarn/hadoop-yarn-applications-distributedshell-2.7.1.jar

-shell\_command uptime

4.4.5 在client机上提交MapReduce型任务

（1）指定在YARN上运行MapReduce任务

首先，在master机上，将文件“/usr/cstor/hadoop/etc/hadoop/mapred-site.xml. template”重命名为“/usr/cstor/hadoop/etc/hadoop/mapred-site.xml”。

接着，编辑此文件并将如下内容嵌入此文件的configuration标签间：

<property><name>mapreduce.framework.name</name><value>yarn</value></property>

最后，将master机的“/usr/local/hadoop/etc/hadoop/mapred-site.xml”文件拷贝到slaveX与client，重新启动集群。

（2）在client端提交PI Estimator任务

首先进入Hadoop安装目录：/usr/cstor/hadoop/，然后提交PI Estimator任务。

命令最后两个两个参数的含义：第一个参数是指要运行map的次数，这里是2次；第二个参数是指每个map任务，取样的个数；而两数相乘即为总的取样数。Pi Estimator使用Monte Carlo方法计算Pi值的，Monte Carlo方法自行百度。

[root@client hadoop]# bin/hadoop jar share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.1.jar pi 2 10

4.5 实验结果

（1）yarn启动之后在master上的web界面上能看到的界面。如图4-3所示：

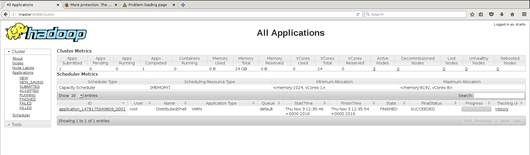


图4-3 web界面总览

（2）提交DistributedShell任务之后web界面看到的界面应该是如图4-4所示：

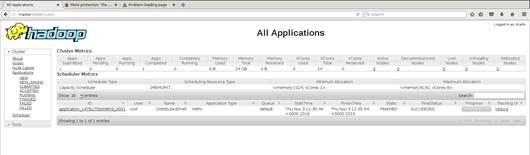


图4-4 DistributedShell任务

（3）提交PI任务之后web界面上看到的如图4-5所示：

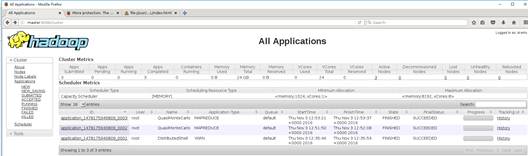


图4-5 MR任务计算PI值

在终端能观察到的界面如图4-6所示：

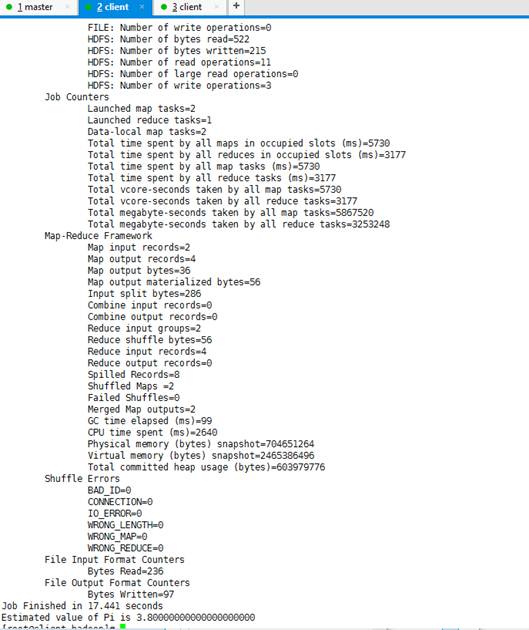


图4-6 MR任务运行结果

实验五 MapReduce实验：单词计数

5.1 实验目的

基于MapReduce思想，编写WordCount程序。

5.2 实验要求

1． 理解MapReduce编程思想；

2． 会编写MapReduce版本WordCount；

3． 会执行该程序；

4． 自行分析执行过程。

5.3 实验原理

MapReduce是一种计算模型，简单的说就是将大批量的工作（数据）分解（MAP）执行，然后再将结果合并成最终结果（REDUCE）。这样做的好处是可以在任务被分解后，可以通过大量机器进行并行计算，减少整个操作的时间。

适用范围：数据量大，但是数据种类小可以放入内存。

基本原理及要点：将数据交给不同的机器去处理，数据划分，结果归约。

理解MapReduce和Yarn：在新版Hadoop中，Yarn作为一个资源管理调度框架，是Hadoop下MapReduce程序运行的生存环境。其实MapRuduce除了可以运行Yarn框架下，也可以运行在诸如Mesos，Corona之类的调度框架上，使用不同的调度框架，需要针对Hadoop做不同的适配。

一个完成的MapReduce程序在Yarn中执行过程如下：

（1）ResourcManager JobClient向ResourcManager提交一个job。

（2）ResourcManager向Scheduler请求一个供MRAppMaster运行的container，然后启动它。

（3）MRAppMaster启动起来后向ResourcManager注册。

（4）ResourcManagerJobClient向ResourcManager获取到MRAppMaster相关的信息，然后直接与MRAppMaster进行通信。

（5）MRAppMaster算splits并为所有的map构造资源请求。

（6）MRAppMaster做一些必要的MR OutputCommitter的准备工作。

（7）MRAppMaster向RM(Scheduler)发起资源请求，得到一组供map/reduce task运行的container，然后与NodeManager一起对每一个container执行一些必要的任务，包括资源本地化等。

（8）MRAppMaster 监视运行着的task 直到完成，当task失败时，申请新的container运行失败的task。

（9）当每个map/reduce task完成后，MRAppMaster运行MR OutputCommitter的cleanup 代码，也就是进行一些收尾工作。

（10）当所有的map/reduce完成后，MRAppMaster运行OutputCommitter的必要的job commit或者abort APIs。

（11）MRAppMaster退出。

5.3.1 MapReduce编程

编写在Hadoop中依赖Yarn框架执行的MapReduce程序，并不需要自己开发MRAppMaster和YARNRunner，因为Hadoop已经默认提供通用的YARNRunner和MRAppMaster程序， 大部分情况下只需要编写相应的Map处理和Reduce处理过程的业务程序即可。

编写一个MapReduce程序并不复杂，关键点在于掌握分布式的编程思想和方法，主要将计算过程分为以下五个步骤：

（1）迭代。遍历输入数据，并将之解析成key/value对。

（2）将输入key/value对映射(map)成另外一些key/value对。

（3）依据key对中间数据进行分组(grouping)。

（4）以组为单位对数据进行归约(reduce)。

（5）迭代。将最终产生的key/value对保存到输出文件中。

5.3.2 Java API解析

（1）InputFormat：用于描述输入数据的格式，常用的为TextInputFormat提供如下两个功能：

数据切分： 按照某个策略将输入数据切分成若干个split，以便确定Map Task个数以及对应的split。

为Mapper提供数据：给定某个split，能将其解析成一个个key/value对。

（2）OutputFormat：用于描述输出数据的格式，它能够将用户提供的key/value对写入特定格式的文件中。

（3）Mapper/Reducer: Mapper/Reducer中封装了应用程序的数据处理逻辑。

（4）Writable:Hadoop自定义的序列化接口。实现该类的接口可以用作MapReduce过程中的value数据使用。

（5）WritableComparable：在Writable基础上继承了Comparable接口，实现该类的接口可以用作MapReduce过程中的key数据使用。（因为key包含了比较排序的操作）。

5.4 实验步骤

本实验主要分为，确认前期准备，编写MapReduce程序，打包提交代码。查看运行结果这几个步骤，详细如下：

5.4.1 启动Hadoop

执行命令启动前面实验部署好的Hadoop系统。

[root@master ~]# cd /usr/cstor/hadoop/

[root@master hadoop]# sbin/start-all.sh

5.4.2 验证HDFS上没有wordcount的文件夹

[root@client ~]# cd /usr/cstor/hadoop/

[root@client hadoop]# bin/hadoop fs -ls / #查看HDFS上根目录文件 /

此时HDFS上应该是没有wordcount文件夹。

5.4.3 上传数据文件到HDFS

[root@client ~]# cd /usr/cstor/hadoop/

[root@client hadoop]# bin/hadoop fs -put /root/data/5/word /

5.4.4 编写MapReduce程序

主要编写Map和Reduce类，其中Map过程需要继承org.apache.hadoop.mapreduce包中Mapper类，并重写其map方法；Reduce过程需要继承org.apache.hadoop.mapreduce包中Reduce类，并重写其reduce方法。

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.Reducer;

import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;

import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

import org.apache.hadoop.mapreduce.lib.partition.HashPartitioner;

import java.io.IOException;

import java.util.StringTokenizer;

public class WordCount {

public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable> {

private final static IntWritable one = new IntWritable(1);

private Text word = new Text();

//map方法，划分一行文本，读一个单词写出一个<单词,1>

public void map(Object key, Text value, Context context)throws IOException, InterruptedException {

StringTokenizer itr = new StringTokenizer(value.toString());

while (itr.hasMoreTokens()) {

word.set(itr.nextToken());

context.write(word, one);//写出<单词,1>

}}}

//定义reduce类，对相同的单词，把它们<K,VList>中的VList值全部相加

public static class IntSumReducer extends Reducer<Text, IntWritable, Text, IntWritable> {

private IntWritable result = new IntWritable();

public void reduce(Text key, Iterable<IntWritable> values,Context context)

throws IOException, InterruptedException {

int sum = 0;

for (IntWritable val : values) {

sum += val.get();//相当于<Hello,1><Hello,1>，将两个1相加

}

result.set(sum);

context.write(key, result);//写出这个单词，和这个单词出现次数<单词，单词出现次数>

}}

public static void main(String[] args) throws Exception {//主方法，函数入口

Configuration conf = new Configuration(); //实例化配置文件类

Job job = new Job(conf, "WordCount"); //实例化Job类

job.setInputFormatClass(TextInputFormat.class); //指定使用默认输入格式类

TextInputFormat.setInputPaths(job, args[0]); //设置待处理文件的位置

job.setJarByClass(WordCount.class); //设置主类名

job.setMapperClass(TokenizerMapper.class); //指定使用上述自定义Map类

job.setCombinerClass(IntSumReducer.class); //指定开启Combiner函数

job.setMapOutputKeyClass(Text.class); //指定Map类输出的<K,V>，K类型

job.setMapOutputValueClass(IntWritable.class); //指定Map类输出的<K,V>，V类型

job.setPartitionerClass(HashPartitioner.class); //指定使用默认的HashPartitioner类

job.setReducerClass(IntSumReducer.class); //指定使用上述自定义Reduce类

job.setNumReduceTasks(Integer.parseInt(args[2])); //指定Reduce个数

job.setOutputKeyClass(Text.class); //指定Reduce类输出的<K,V>,K类型

job.setOutputValueClass(Text.class); //指定Reduce类输出的<K,V>,V类型

job.setOutputFormatClass(TextOutputFormat.class); //指定使用默认输出格式类

TextOutputFormat.setOutputPath(job, new Path(args[1])); //设置输出结果文件位置

System.exit(job.waitForCompletion(true) ? 0 : 1); //提交任务并监控任务状态

}

}

5.4.5 使用Eclipse开发工具将该代码打包

假定打包后的文件名为hdpAction.jar，主类WordCount位于包njupt下，则可使用如下命令向YARN集群提交本应用。

[root@client ~]# ./yarn jar hdpAction.jar njupt.WordCount /word /wordcount 1

其中“yarn”为命令，“jar”为命令参数，后面紧跟打包后的代码地址，“njupt”为包名，“WordCount”为主类名，“/word”为输入文件在HDFS中的位置，/wordcount为输出文件在HDFS中的位置。

5.5 实验结果

5.5.1 程序运行成功控制台上的显示内容

如图5-1所示：



图5-1 提交wordcount

5.5.2 在HDFS上查看结果

[root@client ~]# hdfs dfs –ls /wordcount

[root@client ~]# hdfs dfs –cat /wordcount/part-r-00000

如图5-2所示：



图5-2 查看wordcount结果

实验六 MapReduce实验：二次排序

6.1 实验目的

基于MapReduce思想，编写SecondarySort程序。

6.2 实验要求

要能理解MapReduce编程思想，会编写MapReduce版本二次排序程序，然后将其执行并分析执行过程。

6.3 实验原理

MR默认会对键进行排序，然而有的时候我们也有对值进行排序的需求。满足这种需求一是可以在reduce阶段排序收集过来的values，但是，如果有数量巨大的values可能就会导致内存溢出等问题，这就是二次排序应用的场景——将对值的排序也安排到MR计算过程之中，而不是单独来做。

二次排序就是首先按照第一字段排序，然后再对第一字段相同的行按照第二字段排序，注意不能破坏第一次排序的结果。

6.4 实验步骤

6.4.1 编写程序

程序主要难点在于排序和聚合。

对于排序我们需要定义一个IntPair类用于数据的存储，并在IntPair类内部自定义Comparator类以实现第一字段和第二字段的比较。

对于聚合我们需要定义一个FirstPartitioner类，在FirstPartitioner类内部指定聚合规则为第一字段。

此外，我们还需要开启MapReduce框架自定义Partitioner 功能和GroupingComparator功能。

IntPair 类：

package mr;

import java.io.DataInput;

import java.io.DataOutput;

import java.io.IOException;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.WritableComparable;

public class IntPair implements WritableComparable<IntPair> {

private IntWritable first;

private IntWritable second;

public void set(IntWritable first, IntWritable second) {

this.first = first;

this.second = second;

}

//注意：需要添加无参的构造方法，否则反射时会报错。

public IntPair() {

set(new IntWritable(), new IntWritable());

}

public IntPair(int first, int second) {

set(new IntWritable(first), new IntWritable(second));

}

public IntPair(IntWritable first, IntWritable second) {

set(first, second);

}

public IntWritable getFirst() {

return first;

}

public void setFirst(IntWritable first) {

this.first = first;

}

public IntWritable getSecond() {

return second;

}

public void setSecond(IntWritable second) {

this.second = second;

}

public void write(DataOutput out) throws IOException {

first.write(out);

second.write(out);

}

public void readFields(DataInput in) throws IOException {

first.readFields(in);

second.readFields(in);

}

public int hashCode() {

return first.hashCode() \* 163 + second.hashCode();

}

public boolean equals(Object o) {

if (o instanceof IntPair) {

IntPair tp = (IntPair) o;

return first.equals(tp.first) && second.equals(tp.second);

}

return false;

}

public String toString() {

return first + "\t" + second;

}

public int compareTo(IntPair tp) {

int cmp = first.compareTo(tp.first);

if (cmp != 0) {

return cmp;

}

return second.compareTo(tp.second);

}

}

完整代码：

package mr;

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.NullWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.io.WritableComparable;

import org.apache.hadoop.io.WritableComparator;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.Partitioner;

import org.apache.hadoop.mapreduce.Reducer;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class SecondarySort {

static class TheMapper extends Mapper<LongWritable, Text, IntPair, NullWritable> {

@Override

protected void map(LongWritable key, Text value, Context context)

throws IOException, InterruptedException {

String[] fields = value.toString().split("\t");

int field1 = Integer.parseInt(fields[0]);

int field2 = Integer.parseInt(fields[1]);

context.write(new IntPair(field1,field2), NullWritable.get());

}

}

static class TheReducer extends Reducer<IntPair, NullWritable,IntPair, NullWritable> {

//private static final Text SEPARATOR = new Text("------------------------------------------------");

@Override

protected void reduce(IntPair key, Iterable<NullWritable> values, Context context)

throws IOException, InterruptedException {

context.write(key, NullWritable.get());

}

}

public static class FirstPartitioner extends Partitioner<IntPair, NullWritable> {

public int getPartition(IntPair key, NullWritable value,

int numPartitions) {

return Math.abs(key.getFirst().get()) % numPartitions;

}

}

//如果不添加这个类，默认第一列和第二列都是升序排序的。

//这个类的作用是使第一列升序排序，第二列降序排序

public static class KeyComparator extends WritableComparator {

//无参构造器必须加上，否则报错。

protected KeyComparator() {

super(IntPair.class, true);

}

public int compare(WritableComparable a, WritableComparable b) {

IntPair ip1 = (IntPair) a;

IntPair ip2 = (IntPair) b;

//第一列按升序排序

int cmp = ip1.getFirst().compareTo(ip2.getFirst());

if (cmp != 0) {

return cmp;

}

//在第一列相等的情况下，第二列按倒序排序

return -ip1.getSecond().compareTo(ip2.getSecond());

}

}

//入口程序

public static void main(String[] args) throws Exception {

Configuration conf = new Configuration();

Job job = Job.getInstance(conf);

job.setJarByClass(SecondarySort.class);

//设置Mapper的相关属性

job.setMapperClass(TheMapper.class);

//当Mapper中的输出的key和value的类型和Reduce输出

//的key和value的类型相同时，以下两句可以省略。

//job.setMapOutputKeyClass(IntPair.class);

//job.setMapOutputValueClass(NullWritable.class);

FileInputFormat.setInputPaths(job, new Path(args[0]));

//设置分区的相关属性

job.setPartitionerClass(FirstPartitioner.class);

//在map中对key进行排序

job.setSortComparatorClass(KeyComparator.class);

//job.setGroupingComparatorClass(GroupComparator.class);

//设置Reducer的相关属性

job.setReducerClass(TheReducer.class);

job.setOutputKeyClass(IntPair.class);

job.setOutputValueClass(NullWritable.class);

FileOutputFormat.setOutputPath(job, new Path(args[1]));

//设置Reducer数量

int reduceNum = 1;

if(args.length >= 3 && args[2] != null){

reduceNum = Integer.parseInt(args[2]);

}

job.setNumReduceTasks(reduceNum);

job.waitForCompletion(true);

}

}

6.4.2 打包提交

使用Eclipse开发工具将该代码打包，选择主类为mr.Secondary。如果没有指定主类，那么在执行时就要指定须执行的类。假定打包后的文件名为Secondary.jar，主类SecondarySort位于包mr下，则可使用如下命令向Hadoop集群提交本应用。

[root@client hadoop]# bin/hadoop jar SecondarySort.jar mr.Secondary /user/mapreduce/secsort/in/secsortdata.txt /user/mapreduce/secsort/out 1

其中“hadoop”为命令，“jar”为命令参数，后面紧跟打的包，/user/mapreduce/secsort/in/secsortdata.txt”为输入文件在HDFS中的位置，如果HDFS中没有这个文件，则自己自行上传。“/user/mapreduce/secsort/out/”为输出文件在HDFS中的位置，“1”为Reduce个数。

6.5 实验结果

6.5.1 输入数据

输入数据如下：secsortdata.txt ('\t'分割)（数据放在/root/data/6目录下）：

7 444

3 9999

7 333

4 22

3 7777

7 555

3 6666

6 0

3 8888

4 11

6.5.2 执行结果

在client上执行对hdfs上的文件/user/mapreduce/secsort/out/part-r-00000内容查看的操作

[root@client hadoop]# bin/hadoop fs -cat /user/mapreduce/secsort/out/p\*

如图6-1所示：

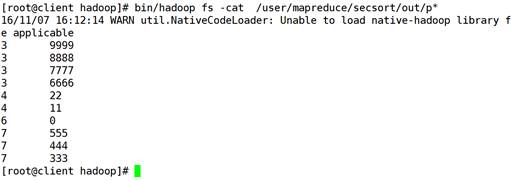


图6-1 查看二次排序结果

实验七 MapReduce实验：计数器

7.1 实验目的

基于MapReduce思想，编写计数器程序。

7.2 实验要求

能够理解MapReduce编程思想，然后会编写MapReduce版本计数器程序，并能执行该程序和分析执行过程。

7.3 实验背景

7.3.1 MapReduce计数器是什么？

计数器是用来记录job的执行进度和状态的。它的作用可以理解为日志。我们可以在程序的某个位置插入计数器，记录数据或者进度的变化情况。

7.3.2 MapReduce计数器能做什么？

MapReduce 计数器（Counter）为我们提供一个窗口，用于观察MapReduce Job运行期的各种细节数据。对MapReduce性能调优很有帮助，MapReduce性能优化的评估大部分都是基于这些Counter 的数值表现出来的。

在许多情况下，一个用户需要了解待分析的数据，尽管这并非所要执行的分析任务的核心内容。以统计数据集中无效记录数目的任务为例，如果发现无效记录的比例相当高，那么就需要认真思考为何存在如此多无效记录。是所采用的检测程序存在缺陷，还是数据集质量确实很低，包含大量无效记录？如果确定是数据集的质量问题，则可能需要扩大数据集的规模，以增大有效记录的比例，从而进行有意义的分析。   
 计数器是一种收集作业统计信息的有效手段，用于质量控制或应用级统计。计数器还可辅助诊断系统故障。如果需要将日志信息传输到map或reduce任务，更好的方法通常是尝试传输计数器值以监测某一特定事件是否发生。对于大型分布式作业而言，使用计数器更为方便。首先，获取计数器值比输出日志更方便，其次，根据计数器值统计特定事件的发生次数要比分析一堆日志文件容易得多。

7.3.3 内置计数器

MapReduce自带了许多默认Counter，现在我们来分析这些默认Counter的含义，方便大家观察Job结果，如输入的字节数、输出的字节数、Map端输入/输出的字节数和条数、Reduce端的输入/输出的字节数和条数等。下面我们只需了解这些内置计数器，知道计数器组名称（groupName）和计数器名称（counterName），以后使用计数器会查找groupName和counterName即可。

7.3.4 计数器使用

1）定义计数器

枚举声明计数器：

// 自定义枚举变量Enum

Counter counter = context.getCounter(Enum enum)

自定义计数器：

// 自己命名groupName和counterName

Counter counter = context.getCounter(String groupName,String counterName)

2）为计数器赋值

初始化计数器：

counter.setValue(long value);//设置初始值

计数器自增：

counter.increment(long incr);// 增加计数

3）获取计数器的值

获取枚举计数器的值：

Configuration conf = new Configuration();

Job job = new Job(conf, "MyCounter");

job.waitForCompletion(true);

Counters counters=job.getCounters();

Counter counter=counters.findCounter(LOG\_PROCESSOR\_COUNTER.BAD\_RECORDS\_LONG);// 查找枚举计数器，假如Enum的变量为BAD\_RECORDS\_LONG

long value=counter.getValue();//获取计数值

获取自定义计数器的值：

Configuration conf = new Configuration();

Job job = new Job(conf, "MyCounter");

job.waitForCompletion(true);

Counters counters = job.getCounters();

Counter counter=counters.findCounter("ErrorCounter","toolong");// 假如groupName为ErrorCounter，counterName为toolong

long value = counter.getValue();// 获取计数值

获取内置计数器的值：

Configuration conf = new Configuration();

Job job = new Job(conf, "MyCounter");

job.waitForCompletion(true);

Counters counters=job.getCounters();

// 查找作业运行启动的reduce个数的计数器，groupName和counterName可以从内置计数器表格查询（前面已经列举有）

Counter counter=counters.findCounter("org.apache.hadoop.mapreduce.JobCounter","TOTAL\_LAUNCHED\_REDUCES") ;// 假如groupName为org.apache.hadoop.mapreduce.JobCounter，counterName为TOTAL\_LAUNCHED\_REDUCES

long value=counter.getValue();// 获取计数值

获取所有计数器的值：

Configuration conf = new Configuration();

Job job = new Job(conf, "MyCounter");

Counters counters = job.getCounters();

for (CounterGroup group : counters) {

for (Counter counter : group) {

System.out.println(counter.getDisplayName() + ": " + counter.getName() + ": "+ counter.getValue());

}

}

7.3.5 自定义计数器

MapReduce允许用户编写程序来定义计数器，计数器的值可在mapper或reducer 中增加。多个计数器由一个Java枚举(enum)类型来定义，以便对计数器分组。一个作业可以定义的枚举类型数量不限，各个枚举类型所包含的字段数量也不限。枚举类型的名称即为组的名称，枚举类型的字段就是计数器名称。计数器是全局的。换言之，MapReduce框架将跨所有map和reduce聚集这些计数器，并在作业结束时产生一个最终结果。

7.4 实验步骤

7.4.1 实验分析设计

该实验要求学生自己实现一个计数器，统计输入的无效数据。说明如下：

假如一个文件，规范的格式是3个字段，“\t”作为分隔符，其中有2条异常数据，一条数据是只有2个字段，一条数据是有4个字段。其内容如下所示：

jim 1 28

kate 0 26

tom 1

lily 0 29 22

编写代码统计文档中字段不为3个的异常数据个数。如果字段超过3个视为过长字段，字段少于3个视为过短字段。

7.4.2 编写程序

完整代码：

package mr ;

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Counter;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import org.apache.hadoop.util.GenericOptionsParser;

public class Counters {

public static class MyCounterMap extends Mapper<LongWritable, Text, Text, Text> {

public static Counter ct = null;

protected void map(LongWritable key, Text value,

org.apache.hadoop.mapreduce.Mapper<LongWritable, Text, Text, Text>.Context context)

throws java.io.IOException, InterruptedException {

String arr\_value[] = value.toString().split("\t");

if (arr\_value.length > 3) {

ct = context.getCounter("ErrorCounter", "toolong"); // ErrorCounter为组名，toolong为组员名

ct.increment(1); // 计数器加一

} else if (arr\_value.length < 3) {

ct = context.getCounter("ErrorCounter", "tooshort");

ct.increment(1);

}

}

}

public static void main(String[] args) throws IOException, InterruptedException, ClassNotFoundException {

Configuration conf = new Configuration();

String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();

if (otherArgs.length != 2) {

System.err.println("Usage: Counters <in> <out>");

System.exit(2);

}

Job job = new Job(conf, "Counter");

job.setJarByClass(Counters.class);

job.setMapperClass(MyCounterMap.class);

FileInputFormat.addInputPath(job, new Path(otherArgs[0]));

FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));

System.exit(job.waitForCompletion(true) ? 0 : 1);

}

}

7.4.3 打包并提交

使用Eclipse开发工具将该代码打包，选择主类为mr.Counters。假定打包后的文件名为Counters.jar，主类Counters位于包mr下，则可使用如下命令向Hadoop集群提交本应用。

[root@client hadoop]# bin/hadoop jar Counters.jar mr.Counters /usr/counters/in/counters.txt /usr/counters/out

其中“hadoop”为命令，“jar”为命令参数，后面紧跟打包。 “/usr/counts/in/counts.txt”为输入文件在HDFS中的位置(如果没有，自行上传)，“/usr/counts/out”为输出文件在HDFS中的位置。

7.5 实验结果

7.5.1 输入数据

输入数据如下：counters.txt （\t分割）（数据统一放在/root/data目录下）。

jim 1 28

kate 0 26

tom 1

lily 0 29 22

7.5.2 输出显示

如图7-1所示：



图7-1 提交计数器输出

实验八 MapReduce实验：Join操作

8.1 实验目的

基于MapReduce思想，编写两文件Join操作的程序。

8.2 实验要求

能够理解MapReduce编程思想，然后会编写MapReduce版本Join程序，并能执行该程序和分析执行过程。

8.3 实验背景

8.3.1 概述

对于RDBMS中的Join操作大伙一定非常熟悉，写SQL的时候要十分注意细节，稍有差池就会耗时巨久造成很大的性能瓶颈，而在Hadoop中使用[MapReduce](http://www.superwu.cn/2014/01/26/989/)框架进行Join的操作时同样耗时，但是由于[Hadoop](http://www.superwu.cn/2014/11/19/1409/)的分布式设计理念的特殊性，因此对于这种Join操作同样也具备了一定的特殊性。

8.3.2 原理

使用MapReduce实现Join操作有多种实现方式：

1）在Reduce端连接为最为常见的模式：

Map端的主要工作：为来自不同表（文件）的key/value对打标签以区别不同来源的记录。然后用连接字段作为key，其余部分和新加的标志作为value，最后进行输出。

Reduce端的主要工作：在Reduce端以连接字段作为key的分组已经完成，我们只需要在每一个分组当中将那些来源于不同文件的记录（在map阶段已经打标志）分开，最后进行笛卡尔积就OK了。

之所以会存在reduce join这种方式，可以很明显的看出原因：因为整体数据被分割了，每个map task只处理一部分数据而不能够获取到所有需要的join字段，因此我们需要再将join key作为reduce端的分组将所有join key相同的记录集中起来进行处理，所以reduce join这种方式就出现了。这种方式的缺点很明显就是会造成map和reduce端连接的shuffle阶段出现大量的数据传输，效率很低。

2）在Map端进行连接

使用场景：一张表十分小、一张表很大。

用法：在提交作业的时候先将小表文件放到该作业的DistributedCache中，然后从DistributeCache中取出该小表进行Join key / value解释分割放到内存中（可以放到Hash Map等容器中）。然后扫描大表，看大表中的每条记录的Join key /value值是否能够在内存中找到相同Join key的记录，如果有则直接输出结果。

3）SemiJoin

SemiJoin就是所谓的半连接，其实仔细一看就是Reduce Join的一个变种，就是在map端过滤掉一些数据，在网络中只传输参与连接的数据不参与连接的数据不必在网络中进行传输，从而减少了shuffle的网络传输量，使整体效率得到提高，其他思想和Reduce Join是一模一样的。说得更加接地气一点就是将小表中参与Join的key单独抽出来通过DistributedCach分发到相关节点，然后将其取出放到内存中（可以放到HashSet中），在map阶段扫描连接表，将Join key不在内存HashSet中的记录过滤掉，让那些参与Join的记录通过shuffle传输到Reduce端进行Join操作，其他的和Reduce Join都是一样的。

8.4 实验步骤

8.4.1 准备阶段

在这里我们介绍最为常见的在Reduce端连接的代码编写流程。

首先准备数据，数据分为两个文件，分别为A表和B表数据：

201001 1003 abc

201002 1005 def

201003 1006 ghi

201004 1003 jkl

201005 1004 mno

201006 1005 pqr

**A表数据**

1003 kaka

1004 da

1005 jue

1006 zhao

**B表数据**

现在要通过程序得到A表第二个字段和B表第一个字段一致的数据的Join结果：

1003 201001 abc kaka

1003 201004 jkl kaka

1004 201005 mno da

1005 201002 def jue

1005 201006 pqr jue

1006 201003 ghi zhao

程序分析执行过程如下：

在map阶段，把所有记录标记成<key, value>的形式，其中key是1003/1004/1005/1006的字段值，value则根据来源不同取不同的形式：来源于表A的记录，value的值为“201001 abc”等值；来源于表B的记录，value的值为“kaka”之类的值。

在Reduce阶段，先把每个key下的value列表拆分为分别来自表A和表B的两部分，分别放入两个向量中。然后遍历两个向量做笛卡尔积，形成一条条最终结果。

8.4.2 编写程序

完整代码：

package mr;

import java.io.DataInput;

import java.io.DataOutput;

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.io.WritableComparable;

import org.apache.hadoop.io.WritableComparator;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.Partitioner;

import org.apache.hadoop.mapreduce.Reducer;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import org.apache.hadoop.mapreduce.lib.input.FileSplit;

import org.apache.hadoop.util.GenericOptionsParser;

public class MRJoin {

public static class MR\_Join\_Mapper extends Mapper<LongWritable, Text, TextPair, Text> {

@Override

protected void map(LongWritable key, Text value, Context context)

throws IOException, InterruptedException {

// 获取输入文件的全路径和名称

String pathName = ((FileSplit) context.getInputSplit()).getPath().toString();

if (pathName.contains("data.txt")) {

String values[] = value.toString().split("\t");

if (values.length < 3) {

// data数据格式不规范，字段小于3，抛弃数据

return;

} else {

// 数据格式规范，区分标识为1

TextPair tp = new TextPair(new Text(values[1]), new Text("1"));

context.write(tp, new Text(values[0] + "\t" + values[2]));

}

}

if (pathName.contains("info.txt")) {

String values[] = value.toString().split("\t");

if (values.length < 2) {

// data数据格式不规范，字段小于2，抛弃数据

return;

} else {

// 数据格式规范，区分标识为0

TextPair tp = new TextPair(new Text(values[0]), new Text("0"));

context.write(tp, new Text(values[1]));

}

}

}

}

public static class MR\_Join\_Partitioner extends Partitioner<TextPair, Text> {

@Override

public int getPartition(TextPair key, Text value, int numParititon) {

return Math.abs(key.getFirst().hashCode() \* 127) % numParititon;

}

}

public static class MR\_Join\_Comparator extends WritableComparator {

public MR\_Join\_Comparator() {

super(TextPair.class, true);

}

public int compare(WritableComparable a, WritableComparable b) {

TextPair t1 = (TextPair) a;

TextPair t2 = (TextPair) b;

return t1.getFirst().compareTo(t2.getFirst());

}

}

public static class MR\_Join\_Reduce extends Reducer<TextPair, Text, Text, Text> {

protected void Reduce(TextPair key, Iterable<Text> values, Context context)

throws IOException, InterruptedException {

Text pid = key.getFirst();

String desc = values.iterator().next().toString();

while (values.iterator().hasNext()) {

context.write(pid, new Text(values.iterator().next().toString() + "\t" + desc));

}

}

}

public static void main(String agrs[])

throws IOException, InterruptedException, ClassNotFoundException {

Configuration conf = new Configuration();

GenericOptionsParser parser = new GenericOptionsParser(conf, agrs);

String[] otherArgs = parser.getRemainingArgs();

if (agrs.length < 3) {

System.err.println("Usage: MRJoin <in\_path\_one> <in\_path\_two> <output>");

System.exit(2);

}

Job job = new Job(conf, "MRJoin");

// 设置运行的job

job.setJarByClass(MRJoin.class);

// 设置Map相关内容

job.setMapperClass(MR\_Join\_Mapper.class);

// 设置Map的输出

job.setMapOutputKeyClass(TextPair.class);

job.setMapOutputValueClass(Text.class);

// 设置partition

job.setPartitionerClass(MR\_Join\_Partitioner.class);

// 在分区之后按照指定的条件分组

job.setGroupingComparatorClass(MR\_Join\_Comparator.class);

// 设置Reduce

job.setReducerClass(MR\_Join\_Reduce.class);

// 设置Reduce的输出

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(Text.class);

// 设置输入和输出的目录

FileInputFormat.addInputPath(job, new Path(otherArgs[0]));

FileInputFormat.addInputPath(job, new Path(otherArgs[1]));

FileOutputFormat.setOutputPath(job, new Path(otherArgs[2]));

// 执行，直到结束就退出

System.exit(job.waitForCompletion(true) ? 0 : 1);

}

}

class TextPair implements WritableComparable<TextPair> {

private Text first;

private Text second;

public TextPair() {

set(new Text(), new Text());

}

public TextPair(String first, String second) {

set(new Text(first), new Text(second));

}

public TextPair(Text first, Text second) {

set(first, second);

}

public void set(Text first, Text second) {

this.first = first;

this.second = second;

}

public Text getFirst() {

return first;

}

public Text getSecond() {

return second;

}

public void write(DataOutput out) throws IOException {

first.write(out);

second.write(out);

}

public void readFields(DataInput in) throws IOException {

first.readFields(in);

second.readFields(in);

}

public int compareTo(TextPair tp) {

int cmp = first.compareTo(tp.first);

if (cmp != 0) {

return cmp;

}

return second.compareTo(tp.second);

}

}

8.4.3 打包并提交

使用Eclipse开发工具将该代码打包，假定打包后的文件名为MRJoin.jar，主类MRJoin位于包mr下，则可使用如下命令向Hadoop集群提交本应用。

[root@client hadoop]# bin/hadoop jar MRJoin.jar mr.MRJoin /usr/MRJoin/in/data.txt /usr/MRJoin/in/info.txt /usr/MRJoin/out

其中“hadoop”为命令，“jar”为命令参数，后面紧跟打包。“/usr/MRJoin/in/data.txt”和“/usr/MRJoin/in/info.txt”为输入文件在HDFS中的位置，“/usr/MRJoin/out”为输出文件在HDFS中的位置。

8.5 实验结果

8.5.1 输入数据

输入数据如下：data.txt （数据统一放在/root/data目录下）。

201001 1003 abc

201002 1005 def

201003 1006 ghi

201004 1003 jkl

201005 1004 mno

201006 1005 pqr

输入数据如下：info.txt （数据统一放在/root/data目录下）。

1003 kaka

1004 da

1005 jue

1006 zhao

8.5.2 输出显示

在client上执行对hdfs上的文件/usr/MRJoin/out/part-r-00000内容查看的操作。

[root@client hadoop]# bin/hadoop fs -cat /usr/MRJoin/out/p\*

如图8-1所示：

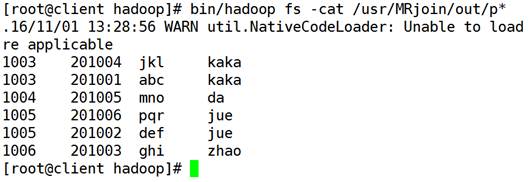


图8-1 查看MR Join结果

实验九 MapReduce实验：分布式缓存

9.1 实验目的

理解序列化与反序列化；熟悉Configuration类；学会使用Configuration类进行参数传递；学会在Map或Reduce阶段引用Configuration传来的参数；理解分布式缓存“加载小表、扫描大表”的处理思想。

9.2 实验要求

假定现有一个大为100G的大表big.txt和一个大小为1M的小表small.txt，请基于MapReduce思想编程实现判断小表中单词在大表中出现次数。也即所谓的“扫描大表、加载小表”。

9.3 实验步骤

为解决上述问题，可开启10个Map、这样，每个Map只需处理总量的1/10，将大大加快处理。而在单独Map内，直接用HashSet加载“1M小表”，对于存在硬盘（Map处理时会将HDFS文件拷贝至本地）的10G大文件，则逐条扫描，这就是所谓的“扫描大表、加载小表”，也即分布式缓存，如图9-1所示。

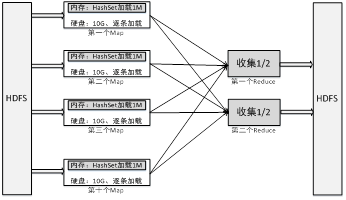


图9-1 加载小表扫描大表示意图

由于实验中没有100G这样的大表，甚至连1M这样的小表都没有，因为本实验采用模拟方式，用少量数据代表大文件big.txt，更少量数据代表small.txt。整个实验步骤为“准备数据上传数据编写代码执行代码查看结果”这五大步骤。

9.3.1 准备数据

为降低操作难度，此处用少量数据代表大文件big.txt，更少量数据代表小文件small.txt，具体操作如下：

首先，登录client机，确认该机上存在“/root/data/9/big.txt”和“/root/data/9/ small.txt”，如图9-2所示，显然big.txt内容为“aaa~zzz和000~999”，small.txt为其中三项。

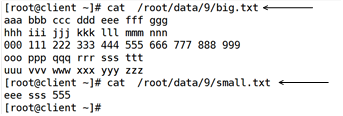


图9-2 确认本地文件big.txt和small.txt

9.3.2 上传数据

首先，登录client机，查看HDFS里是否已存在目录“/user/root/mr/in”，若不存在，使用下述命令新建该目录。

[root@client ~]# /usr/cstor/hadoop/bin/hdfs dfs -mkdir -p /user/root/mr/in

接着，使用下述命令将client机本地文件“/root/data/9/big.txt”和“/root/data/9/ small.txt”上传至HDFS的“/user/root/mr/in”目录：

[root@client ~]# /usr/cstor/hadoop/bin/hdfs dfs -put /root/data/9/big.txt /user/root/mr/in

[root@client ~]# /usr/cstor/hadoop/bin/hdfs dfs -put /root/data/9/small.txt /user/root/mr/in

最后，使用下述命令确认HDFS上文件与内容，如图9-3所示。

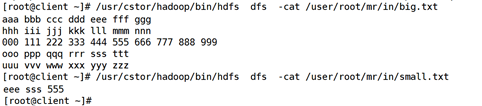


图9-3 确认HDFS文件big.txt和small.txt

9.3.3 编写代码

首先，打开Eclipse，依次点击“FileNewOther…Map/Reduce Project”，在弹出的“New MapReduce Project Wizard”对话框中，“Project name:”一栏填写项目名“BigSmallTable”，然后直接点击该对话框的“Finish”按钮，如图9-4所示。

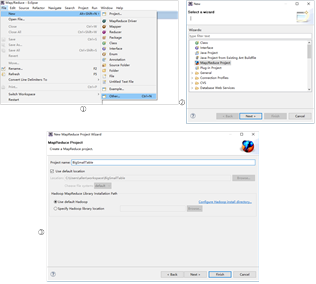


图9-4 确认文件big.txt和small.txt

接着，新建BigAndSmallTable类并指定包名（代码中为cn.cstor.mr），在BigAndSmallTable.java文件中，依次写入如下代码：

package cn.cstor.mr;

import java.io.IOException;

import java.util.HashSet;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.FSDataInputStream;

import org.apache.hadoop.fs.FileSystem;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.Reducer;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import org.apache.hadoop.util.LineReader;

public class BigAndSmallTable {

public static class TokenizerMapper extends

Mapper<Object, Text, Text, IntWritable> {

private final static IntWritable one = new IntWritable(1);

private static HashSet<String> smallTable = null;

protected void setup(Context context) throws IOException,

InterruptedException {

smallTable = new HashSet<String>();

Path smallTablePath = new Path(context.getConfiguration().get(

"smallTableLocation"));

FileSystem hdfs = smallTablePath.getFileSystem(context

.getConfiguration());

FSDataInputStream hdfsReader = hdfs.open(smallTablePath);

Text line = new Text();

LineReader lineReader = new LineReader(hdfsReader);

while (lineReader.readLine(line) > 0) {

// you can do something here

String[] values = line.toString().split(" ");

for (int i = 0; i < values.length; i++) {

smallTable.add(values[i]);

System.out.println(values[i]);

}

}

lineReader.close();

hdfsReader.close();

System.out.println("setup ok \*^\_^\* ");

}

public void map(Object key, Text value, Context context)

throws IOException, InterruptedException {

String[] values = value.toString().split(" ");

for (int i = 0; i < values.length; i++) {

if (smallTable.contains(values[i])) {

context.write(new Text(values[i]), one);

}

}

}

}

public static class IntSumReducer extends

Reducer<Text, IntWritable, Text, IntWritable> {

private IntWritable result = new IntWritable();

public void reduce(Text key, Iterable<IntWritable> values,

Context context) throws IOException, InterruptedException {

int sum = 0;

for (IntWritable val : values) {

sum += val.get();

}

result.set(sum);

context.write(key, result);

}

}

public static void main(String[] args) throws Exception {

Configuration conf = new Configuration();

conf.set("smallTableLocation", args[1]);

Job job = Job.getInstance(conf, "BigAndSmallTable");

job.setJarByClass(BigAndSmallTable.class);

job.setMapperClass(TokenizerMapper.class);

job.setReducerClass(IntSumReducer.class);

job.setMapOutputKeyClass(Text.class);

job.setMapOutputValueClass(IntWritable.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(IntWritable.class);

FileInputFormat.addInputPath(job, new Path(args[0]));

FileOutputFormat.setOutputPath(job, new Path(args[2]));

System.exit(job.waitForCompletion(true) ? 0 : 1);

}

}

图9-5位本项目结构图，请读者对照该图，分析项目结构图中各模块。

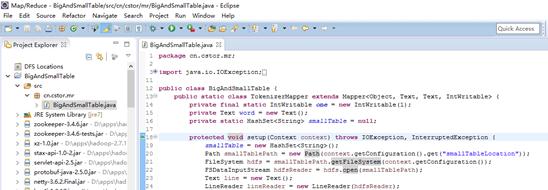


图9-5 项目开发过程示例

待代码编写结束，选中该项目，依次点击“ExportJavaJAR file”，弹出对话框如图9-6中填写打包位置，接着Finish即可。笔者此处打包时包名及其位置为“C:\Users\allen\Desktop\BigSmallTable.jar”。



图9-6 项目开发过程示例

9.3.4 执行代码

首先，使用“Xmanager Enterprise 5”将“C:\Users\allen\ Desktop\BigSmallTable.jar”上传至client机。此处上传至“/root/BigSmallTable.jar”

接着，登录client机上，使用下述命令提交BigSmallTable.jar任务。

[root@client ~]# /usr/cstor/hadoop/bin/hadoop jar /root/BigSmallTable.jar

cn.cstor.mr.BigAndSmallTable /user/root/mr/in/big.txt

/user/root/mr/in/small.txt /user/root/mr/bigAndSmallResult

9.3.5 查看结果

程序执行后，可使用下述命令查看执行结果，注意若再次执行，请更改结果目录：

[root@client ~]# /usr/cstor/hadoop/bin/hdfs dfs -cat /user/root/mr/bigAndSmallResult/part-r-00000

9.4 实验结果

实验结果如图9-7所示，根据big.txt，small.txt文件内容和编程目的，易知实验结果准确无误。

https://bd.cstor.cn/experiment/exp9.files/image007.jpg

图9-7 查看实验结果

实验十 Hive实验：部署Hive

10.1 实验目的

1. 理解Hive存在的原因；

2. 理解Hive的工作原理；

3. 理解Hive的体系架构；

4. 并学会如何进行内嵌模式部署；

5. 启动Hive，然后将元数据存储在HDFS上。

10.2 实验要求

1. 完成Hive的内嵌模式部署；

2. 能够将Hive数据存储在HDFS上；

3. 待Hive环境搭建好后，能够启动并执行一般命令。

10.3 实验原理

Hive是Hadoop 大数据生态圈中的数据仓库，其提供以表格的方式来组织与管理HDFS上的数据、以类SQL的方式来操作表格里的数据，Hive的设计目的是能够以类SQL的方式查询存放在HDFS上的大规模数据集，不必开发专门的MapReduce应用。

Hive本质上相当于一个MapReduce和HDFS的翻译终端，用户提交Hive脚本后，Hive运行时环境会将这些脚本翻译成MapReduce和HDFS操作并向集群提交这些操作。

当用户向Hive提交其编写的HiveQL后，首先，Hive运行时环境会将这些脚本翻译成MapReduce和HDFS操作，紧接着，Hive运行时环境使用Hadoop命令行接口向Hadoop集群提交这些MapReduce和HDFS操作，最后，Hadoop集群逐步执行这些MapReduce和HDFS操作，整个过程可概括如下：

（1）用户编写HiveQL并向Hive运行时环境提交该HiveQL。

（2）Hive运行时环境将该HiveQL翻译成MapReduce和HDFS操作。

（3）Hive运行时环境调用Hadoop命令行接口或程序接口，向Hadoop集群提交翻译后的HiveQL。

（4）Hadoop集群执行HiveQL翻译后的MapReduce-APP或HDFS-APP。

由上述执行过程可知，Hive的核心是其运行时环境，该环境能够将类SQL语句编译成MapReduce。

Hive构建在基于静态批处理的Hadoop 之上，Hadoop 通常都有较高的延迟并且在作业提交和调度的时候需要大量的开销。因此，Hive 并不能够在大规模数据集上实现低延迟快速的查询，例如，Hive 在几百MB 的数据集上执行查询一般有分钟级的时间延迟。

因此，Hive 并不适合那些需要低延迟的应用，例如，联机事务处理（OLTP）。Hive 查询操作过程严格遵守Hadoop MapReduce 的作业执行模型，Hive 将用户的HiveQL 语句通过解释器转换为MapReduce 作业提交到Hadoop 集群上，Hadoop 监控作业执行过程，然后返回作业执行结果给用户。Hive 并非为联机事务处理而设计，Hive 并不提供实时的查询和基于行级的数据更新操作。Hive 的最佳使用场合是大数据集的批处理作业，例如，网络日志分析。

Hive架构与基本组成如图10-1所示：

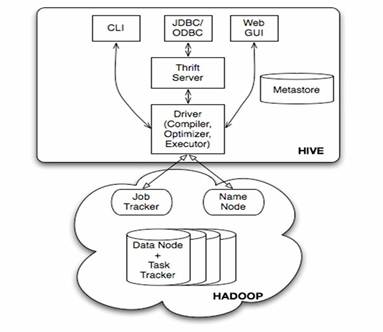


图10-1 Hive体系结构

10.4 实验步骤

相对于其他组件，Hive部署要复杂得多，按metastore存储位置的不同，其部署模式分为内嵌模式、本地模式和完全远程模式三种。当使用完全模式时，可以提供很多用户同时访问并操作Hive，并且此模式还提供各类接口（BeeLine，CLI，甚至是Pig），这里我们以内嵌模式为例。

由于使用内嵌模式时，其Hive会使用内置的Derby数据库来存储数据库，此时无须考虑数据库部署连接问题，整个部署过程可概括如下。

10.4.1 安装部署

在client机上操作：首先确定存在Hive

[root@client~]# ls /usr/cstor/

hive/

[root @client~]#

10.4.2 配置HDFS

先为Hive配置Hadoop安装路径。

待解压完成后，进入Hive的配置文件夹conf目录下，接着将Hive的环境变量模板文件复制成环境变量文件。

[root@client~]# cd /usr/cstor/hive/conf

[root@client conf]# cp hive-env.sh.template hive-env.sh

[root@client conf]# vim hive-env.sh

在配置文件中加入以下语句：

HADOOP\_HOME=/usr/cstor/hadoop

然后在HDFS里新建Hive的存储目录。

在HDFS中新建/tmp 和 /usr/hive/warehouse 两个文件目录，并对同组用户增加写权限。

[root@client hadoop]# bin/hadoop fs -mkdir /tmp

[root@client hadoop]# bin/hadoop fs -mkdir -p /usr/hive/warehouse

[root@client hadoop]# bin/hadoop fs -chmod g+w /tmp

[root@client hadoop]# bin/hadoop fs -chmod g+w /usr/hive/warehouse

10.4.3 启动Hive

在内嵌模式下，启动Hive指的是启动Hive运行时环境，用户可使用下述命令进入Hive运行时环境。

启动Hive命令行：

[root@client ~]# cd /usr/cstor/hive/

[root@client hive ]# bin/hive

10.5 实验结果

10.5.1 启动结果

使用“bin/Hive”命令进入Hive环境验证Hive是否启动成功。如图10-2所示：

[root@client Hive ]# bin/Hive

https://bd.cstor.cn/experiment/exp10.files/image002.jpg

图10-2 启动Hive

10.5.2 Hive基本命令

使用 “bin/Hive” 命令进入Hive环境后，使用 “show tables”， “show function”后如下图所示则表示配置成功。

Hive> show tables ;

如图10-3所示：

https://bd.cstor.cn/experiment/exp10.files/image003.jpg

图10-3 show tables结果

Hive> show functions ;

如图10-4所示：



图10-4 show functions结果

实验十一 Hive实验：新建Hive表

11.1 实验目的

1. 学会创建Hive的表；

2. 显示Hive中的所有表；

3. 显示Hive中表的列项；

4. 修改Hive中的表并能够删除Hive中的表。

11.2 实验要求

1. 要求实验结束时；

2. 每位学生均能够完成Hive的DDL操作；

3. 能够在Hive中新建，显示，修改和删除表等功能。

11.3 实验原理

Hive没有专门的数据存储格式，也没有为数据建立索引，用户可以非常自由的组织 Hive中的表，只需要在创建表的时候告诉Hive数据中的列分隔符和行分隔符，Hive就可以解析数据。

Hive中所有的数据都存储在HDFS中，Hive中包含以下数据模型：表(Table)，外部表(External Table)，分区(Partition)，桶(Bucket)。

Hive中Table和数据库中 Table在概念上是类似的，每一个Table在Hive中都有一个相应的目录存储数据。例如，一个表 pvs，它在 HDFS 中的路径为：/wh/pvs，其中，wh 是在hive-site.xml中由${hive.metastore.warehouse.dir}指定的数据仓库的目录，所有的 Table 数据（不包括 External Table）都保存在这个目录中。

11.4 实验步骤

11.4.1 启动Hive

启动Hive命令行。

[root@client ~]# cd /usr/cstor/hive/

[root@client hive ]# bin/hive

11.4.2 创建表

默认情况下，新建表的存储格式均为Text类型，字段间默认分隔符为键盘上的Tab键。

创建一个有两个字段的pokes表，其中第一列名为foo，数据类型为INT，第二列名为bar，类型为STRING。

hive> CREATE TABLE pokes (foo INT, bar STRING) ;

创建一个有两个实体列和一个（虚拟）分区字段的invites表。

hive> CREATE TABLE invites (foo INT, bar STRING) PARTITIONED BY (ds STRING) ;

注意：分区字段并不属于invites，当向invites导入数据时，ds字段会用来过滤导入的数据。

11.4.3 显示表

显示所有的表。

hive> SHOW TABLES ;

显示表（正则查询），同MySQL中操作一样，Hive也支持正则查询，比如显示以.s结尾的表。

hive> SHOW TABLES '.\*s';

11.4.4 显示表列

hive> DESCRIBE invites;

11.4.5 更改表

修改表events名为3koobecaf (自行创建任意类型events表)：

hive> ALTER TABLE events RENAME TO 3koobecaf;

将pokes表新增一列（列名为new\_col，类型为INT）：

hive> ALTER TABLE pokes ADD COLUMNS (new\_col INT);

将invites表新增一列（列名为new\_col2，类型为INT），同时增加注释“a comment”：

hive> ALTER TABLE invites ADD COLUMNS (new\_col2 INT COMMENT 'a comment');

替换invites表所有列名（数据不动）：

hive> ALTER TABLE invites REPLACE COLUMNS (foo INT, bar STRING, baz INT COMMENT 'baz replaces new\_col2');

11.4.6 删除表（或列）

删除invites表bar 和 baz 两列：

hive> ALTER TABLE invites REPLACE COLUMNS (foo INT COMMENT 'only keep the first column');

删除pokes表：

hive> DROP TABLE pokes;

11.5 实验结果

上述实验步骤的结果如下图所示：

创建一个有两个字段的pokes表，如图11-1所示。

https://bd.cstor.cn/experiment/exp11.files/image001.jpg

图11-1 创建pokes表

创建一个有两个实体列和一个（虚拟）分区字段的invites表，如图11-2所示。

https://bd.cstor.cn/experiment/exp11.files/image002.jpg

图11-2 创建invites表

显示所有的表，如图10-3所示。

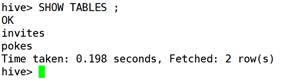


图11-3 显示所有表

以s结尾的表，如图11-4所示。

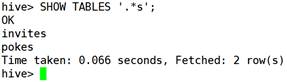


图11-4 显示所有以s结尾的表

显示表列，如图11-5所示。

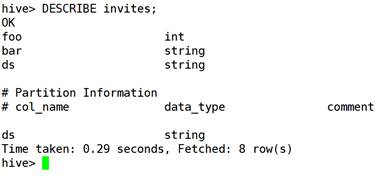


图11-5 描述表

修改表events名为3koobecaf (自行创建任意类型events表)，如图11-6所示。

https://bd.cstor.cn/experiment/exp11.files/image006.jpg

图11-6 表重命名

将pokes表新增一列（列名为new\_col，类型为INT），如图11-7所示。

https://bd.cstor.cn/experiment/exp11.files/image007.jpg

图11-7 表pokes新增列

将invites表新增一列（列名为new\_col2，类型为INT），如图11-8所示。

https://bd.cstor.cn/experiment/exp11.files/image008.jpg

图11-8 表invites新增列

替换invites表所有列名，如图11-9所示。

https://bd.cstor.cn/experiment/exp11.files/image009.jpg

图11-9 替换invites表的列名

删除invites表bar 和 baz 两列，如图11-10所示。

https://bd.cstor.cn/experiment/exp11.files/image010.jpg

图11-10 删除表列

删除pokes表，如图11-11所示。

https://bd.cstor.cn/experiment/exp11.files/image011.jpg

图11-11 删除表

实验十二 Hive实验：Hive分区

12.1 实验目的

掌握Hive分区的用法，加深对Hive分区概念的理解，了解Hive表在HDFS的存储目录结构。

12.2 实验要求

创建一个Hive分区表；根据数据年份创建year=2014和year=2015两个分区；将2015年的数据导入到year=2015的分区；在Hive界面用条件year=2015查询2015年的数据。

12.3 实验原理

分区(Partition) 对应于数据库中的 分区(Partition) 列的密集索引，但是 Hive 中 分区(Partition) 的组织方式和数据库中的很不相同。在 Hive 中，表中的一个分区(Partition) 对应于表下的一个目录，所有的分区(Partition) 的数据都存储在对应的目录中。例如：pvs 表中包含 ds 和 ctry 两个分区(Partition)，则对应于 ds = 20090801, ctry = US 的 HDFS 子目录为：/wh/pvs/ds=20090801/ctry=US；对应于 ds = 20090801, ctry = CA 的 HDFS 子目录为；/wh/pvs/ds=20090801/ctry=CA。

外部表(External Table) 指向已经在 HDFS 中存在的数据，可以创建分区(Partition)。它和 Table 在元数据的组织上是相同的，而实际数据的存储则有较大的差异。

Table 的创建过程和数据加载过程（这两个过程可以在同一个语句中完成），在加载数据的过程中，实际数据会被移动到数据仓库目录中；之后对数据的访问将会直接在数据仓库目录中完成。删除表时，表中的数据和元数据将会被同时删除。

12.4 实验步骤

因为Hive依赖于MapReduce，所以本实验之前先要启动Hadoop集群，然后再启动Hive进行实验，主要包括以下三个步骤。

12.4.1 启动Hadoop集群

在主节点进入Hadoop安装目录，启动Hadoop集群。

[root@master ~]# cd /usr/cstor/hadoop/sbin

[root@master sbin]# ./start-all.sh

12.4.2 用命令进入Hive客户端

进入Hive安装目录，用命令进入Hive客户端。

[root@master ~]# cd /usr/cstor/hive

[root@master hive]# bin/hive

12.4.3 通过HQL语句进行实验

进入客户端后，查看Hive数据库，并选择default数据库：

hive> show databases;  
 OK  
 default  
 Time taken: 1.152 seconds, Fetched: 1 row(s)  
 hive> use default;  
 OK  
 Time taken: 0.036 seconds

在命令端创建Hive分区表：

hive> create table parthive (createdate string, value string) partitioned by (year string) row format delimited

fields terminated by '\t';  
 OK  
 Time taken: 0.298 seconds

查看新建的表：

hive> show tables;  
 OK  
 parthive  
 Time taken: 1.127 seconds, Fetched: 1 row(s)

给parthive表创建两个分区：

hive> alter table parthive add partition(year='2014');  
 OK  
 Time taken: 0.195 seconds  
 hive> alter table parthive add partition(year='2015');  
 OK  
 Time taken: 0.121 seconds

查看parthive的表结构：

hive> describe parthive;

OK

createdate string

value string

year string

# Partition Information

# col\_name data\_type comment

year string

Time taken: 0.423 seconds, Fetched: 8 row(s)

向year=2015分区导入本地数据：

hive> load data local inpath '/root/data/12/parthive.txt' into table parthive partition(year='2015');  
 Loading data to table default.parthive partition (year=2015)

Partition default.parthive{year=2015} stats: [numFiles=1, totalSize=110]  
 OK  
 Time taken: 1.071 seconds

根据条件查询year=2015的数据：

hive> select \* from parthive t where t.year='2015';

根据条件统计year=2015的数据：

hive> select count(\*) from parthive where year='2015';

12.5 实验结果

用命令查看HDFS文件，Hive中parthive表在HDFS文件中的存储目录结构如下图12-1所示：

https://bd.cstor.cn/experiment/exp12.files/image001.jpg

图12-1 parthive表结构图

Hive客户端查询结果如下图12-2所示：



图12-2 客户端查询结果图

Hive客户端统计结果如下图12-3所示：

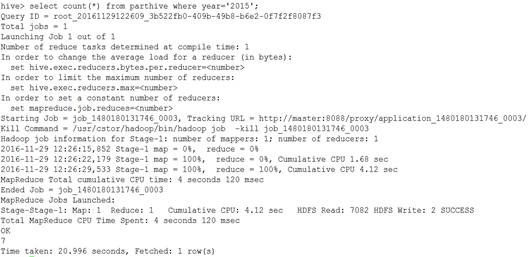


图12-3 客户端统计结果图

实验十三 Spark实验：部署Spark集群

13.1 实验目的

能够理解Spark存在的原因，了解Spark的生态圈，理解Spark体系架构并理解Spark计算模型。学会部署Spark集群并启动Spark集群，能够配置Spark集群使用HDFS。

13.2 实验要求

要求实验结束时，每位学生均已构建出Spark集群：master上部署主服务Master；slave1、2、3上部署从服务Worker；client上部署Spark客户端。待集群搭建好后，还需在client上进行下述操作：提交并运行Spark示例代码WordCount，将client上某文件上传至HDFS刚才新建的目录中。

13.3 实验原理

13.3.1 Spark简介

Spark是一个高速的通用型集群计算框架，其内部内嵌了一个用于执行DAG（有向无环图）的工作流引擎，能够将DAG类型的Spark-App拆分成Task序列并在底层框架上运行。在程序接口层，Spark为当前主流语言都提供了编程接口，如用户可以使用Scala、Java、Python、R等高级语言直接编写Spark-App。此外，在核心层之上，Spark还提供了诸如SQL、Mllib、GraphX、Streaming等专用组件，这些组件内置了大量专用算法，充分利用这些组件，能够大大加快Spark-App开发进度。

一般称Spark Core为Spark，Spark Core处于存储层和高层组建层之间，定位为计算引擎，核心功能是并行化执行用户提交的DAG型Spark-App。目前，Spark生态圈主要包括Spark Core和基于Spark Core的独立组件（SQL、Streaming、Mllib和Graphx）。

13.3.2 Spark适用场景

（1）Spark是基于内存的迭代计算框架，适用于需要多次操作特定数据集的应用场合。

（2）由于RDD的特性，Spark不适用那种异步细粒度更新状态的应用，例如web服务的存储或者是增量的web爬虫和索引。

（3）数据量不是特别大，但是要求实时统计分析需求。

13.4 实验内容和步骤

13.4.1 配置Spark集群

配置Spark集群（独立模式）：

前提：1、请自行配置各节点之间的免密登录，并在/etc/hosts中写好hostname与IP的对应，这样方便配置文件的相互拷贝。2、因为下面实验涉及Spark集群使用HDFS，所以按照之前的实验预先部署好HDFS。

在master机上操作：确定存在spark。

[root@master ~]# ls /usr/cstor

spark/

[root@master ~]#

在master机上操作：进入/usr/cstor目录中。

[root@master ~]# cd /usr/cstor

[root@master cstor]#

进入配置文件目录/usr/cstor/spark/conf， 先拷贝并修改slave.template为slaves。

[root@master ~]# cd /usr/cstor/spark/conf

[root@master cstor]# cp slaves.template slaves

然后用vim命令编辑器编辑slaves文件

[root@master cstor]# vim slaves

编辑slaves文件将下述内容添加到slaves文件中。

slave1

slave2

slave3

上述内容表示当前的Spark集群共有三台slave机，这三台机器的机器名称分别是slave1~3。

在spark-config.sh中加入JAVA\_HOME。

[root@master cstor]# vim /usr/cstor/spark/sbin/spark-config.sh

加入以下内容

export JAVA\_HOME=/usr/local/jdk1.7.0\_79

将配置好的Spark拷贝至slaveX、client。（machines在目录/root/data/2下，如果不存在则自己新建一个）

使用for循环语句完成多机拷贝。

[root@master ~]# cd /root/data/2

[root@master ~]# cat machines

slave1

slave2

slave3

client

[root@master ~]# for x in `cat machines` ; do echo $x ; scp -r /usr/cstor/spark/ $x:/usr/cstor/; done;

在master机上操作：启动Spark集群。

[root@master local]# /usr/cstor/spark/sbin/start-all.sh

13.4.2 配置HDFS

配置Spark集群使用HDFS：

首先关闭集群(在master上执行)

[root@master ~]# /usr/cstor/spark/sbin/stop-all.sh

将Spark环境变量模板复制成环境变量文件。

[root@master ~]# cd /usr/cstor/spark/conf

[root@master conf]# cp spark-env.sh.template spark-env.sh

修改Spark环境变量配置文件spark-env.sh。

[root@master conf]$ vim spark-env.sh

在sprak-env.sh配置文件中添加下列内容。

export HADOOP\_CONF\_DIR=/usr/cstor/hadoop/etc/hadoop

重新启动spark

[root@master local]# /usr/cstor/spark/sbin/start-all.sh

13.4.3 提交Spark任务

在client机上操作：使用Shell命令向Spark集群提交Spark-App

（1）上传in.txt文件到HDFS（hdfs://master:8020/user/spark/in/）上。

in.txt文件在/root/data/13/目录下。

请大家参照实验一自行完成。

（2）提交wordcount 示例代码。

进入/usr/cstor/spark目录，执行如下命令：

bin/spark-submit --master spark://master:7077 --class org.apache.spark.examples.JavaWordCount lib/spark-examples-1.6.0-hadoop2.6.0.jar hdfs://master:8020/user/spark/in/in.txt

[root@master local]# cd /usr/cstor/spark

[root@client spark]# bin/spark-submit --master spark://master:7077 \

> --class org.apache.spark.examples.JavaWordCount \

> lib/spark-examples-1.6.0-hadoop2.6.0.jar hdfs://master:8020/user/spark/in/in.txt

13.5 实验结果

13.5.1 进程查看

在master和slave1-3上分别执行jps命令查看对应进程。master中进程为Master，slave机进程为Worker。如图13-1所示：

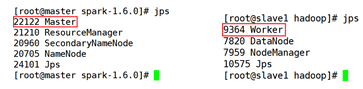


图13-1 查看进程

13.5.2 验证WebUI

在本地（需开启OpenVPN）浏览器中输入master的IP和端口号8080（例：10.1.89.5：8080），即可看到Spark的WebUI。此页面包含了Spark集群主节点、从节点等各类统计信息。如图13-2所示：

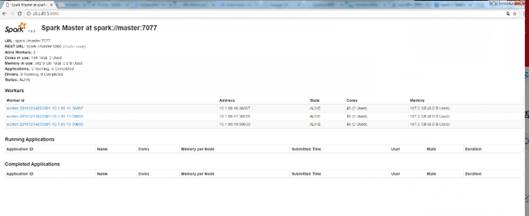


图13-2 spark-web页面

13.5.3 SparkWordcount程序执行

输入：in.txt（数据放在/root/date/13目录下）。

hello world

ni hao

hello my friend

ni are my sunshine

输出结果如图13-3所示

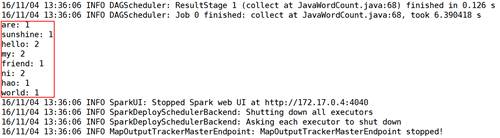


图13-3 输出结果

WebUI中Application的详细信息如图13-4所示。

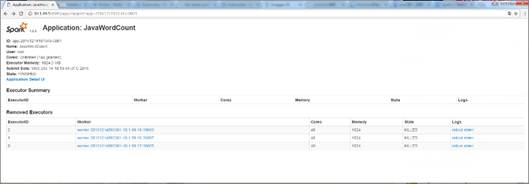


图13-4 web中显示信息

实验十四 Spark实验：SparkWordCount

14.1 实验目的

熟悉Scala语言，基于Spark思想，编写SparkWordCount程序。

14.2 实验要求

熟悉Scala语言，理解Spark编程思想，并会编写Spark 版本的WordCount，然后能够在spark-shell中执行代码和分析执行过程。

14.3 实验原理

Scala 是一门以 Java 虚拟机（JVM）为目标运行环境并将面向对象 (OO) 和函数式编程语言 (FP) 的最佳特性结合在一起的编程语言。

它既有动态语言那样的灵活简洁，同时又保留了静态类型检查带来的安全保障和执行效率，加上其强大的抽象能力，既能处理脚本化的临时任务，又能处理高并发场景下的分布式互联网大数据应用，可谓能缩能伸。

Scala 运行在JVM之上，因此它可以访问任何 Java 类库并且与 Java 框架进行互操作。其与Java的集成度很高，可以直接使用 Java 社区大量成熟的技术框架和方案。由于它直接编译成Java字节码，因此我们可以充分利用JVM这个高性能的运行平台。

14.3.1 Scala是兼容的

Scala被设计成无缝地与Java实施互操作。不需要你从Java平台后退两步然后跳到Java语言前面去。它允许你在现存代码中加点儿东西——在你已有的东西上建设，Scala程序会被编译为JVM的字节码。它们的执行期性能通常与Java程序一致。Scala代码可以调用Java方法，访问Java字段，继承自Java类和实现Java接口。这些都不需要特别的语法，显式接口描述，或粘接代码。实际上，几乎所有Scala代码都极度依赖于Java库，而程序员无须意识到这点。

交互式操作的另一个方面是Scala极度重用了Java类型 。Scala的Int类型代表了Java的原始整数类型int，Float代表了float，Boolean代表boolean，等等。Scala的数组被映射到Java数组。Scala同样重用了许多标准Java库类型。例如，Scala里的字串文本”abc”是java.lang.String，而抛出的异常必须是java.lang.Throwable的子类。

Scala不仅重用了Java的类型，还把它们“打扮”得更漂亮 。例如，Scala的字串支持类似于toInt和toFloat的方法，可以把字串转换成整数或者浮点数。因此你可以写str.toInt替代Integer.parseInt(str)。如何在不打破互操作性的基础上做到这点呢？Java的String类当然不会有toInt方法。实际上，Scala有一个解决这种高级库设计和互操作性不相和谐的通用方案。Scala可以让你定义隐式转换：implicit conversion，这常常用在类型失配，或者选用不存在的方法时。在上面的例子里，当在字串中寻找toInt方法时，Scala编译器会发现String类里没有这种方法，但它会发现一个把Java的String转换为Scala的RichString类的一个实例的隐式转换，里面定义了这么个方法。于是在执行toInt操作之前，转换被隐式应用。

Scala代码同样可以由Java代码调用。有时这种情况要更加微妙，因为Scala是一种比Java更丰富的语言，有些Scala更先进的特性在它们能映射到Java前需要先被编码一下。

14.3.2 Scala是简洁的

Scala程序一般都很短。Scala程序员曾报告说与Java比起来代码行数可以减少到1/10。这有可能是个极限的例子。较保守的估计大概标准的Scala程序应该有Java写的同样的程序一半行数左右。更少的行数不仅意味着更少的打字工作，同样意味着更少的话在阅读和理解程序上的努力及更少的出错可能。许多因素在减少代码行上起了作用。Scala的语法避免了一些束缚Java程序的固定写法。例如，Scala里的分号是可选的，且通常不写。Scala语法里还有很多其他的地方省略了东西。比方说，比较一下你在Java和Scala里是如何写类及构造函数的。

在Java里，带有构造函数的类经常看上去是这个样子：

// 在Java里

class MyClass {

private int index;

private String name;

public MyClass(int index, String name) {

this.index = index;

this.name = name;

}

}

在Scala里，你会写成这样：

　　class MyClass(index: Int, name: String)

根据这段代码，Scala编译器将制造有两个私有成员变量的类，一个名为index的Int类型和一个叫做name的String类型，还有一个用这些变量作为参数获得初始值的构造函数。这个构造函数还将用作为参数传入的值初始化这两个成员变量。Scala类写起来更快，读起来更容易，最重要的是，比Java类更不容易犯错。

有助于Scala的简洁易懂的另一个因素是它的类型推断。重复的类型信息可以被忽略，因此程序变得更有条理和易读，但或许减少代码最关键的是因为已经存在于你的库里而不需要写的代码。Scala给了你许多工具来定义强有力的库让你抓住并提炼出通用的行为。例如，库类的不同方面可以被分成若干特质，而这些又可以被灵活地混合在一起。或者，库方法可以用操作符参数化，从而让你有效地定义那些你自己控制的构造。这些构造组合在一起，就能够让库的定义既是高层级的又能灵活运用。

14.3.3 Scala是高级的

程序员总是在和复杂性纠缠。为了高产出的编程，你必须明白你工作的代码。过度复杂的代码成了很多软件工程崩溃的原因。不幸的是，重要的软件往往有复杂的需求。这种复杂性不可避免；必须（由不受控）转为受控。

Scala可以通过让你提升你设计和使用的接口的抽象级别来帮助你管理复杂性。例如，假设你有一个String变量name，你想弄清楚是否String包含一个大写字符。

在Java里，你或许这么写：

// 在Java里

boolean nameHasUpperCase = false;

for (int i = 0; i < name.length(); ++i) {

if (Character.isUpperCase(name.charAt(i))) {

nameHasUpperCase = true;

break;

}

}

在Scala里，你可以写成：

val nameHasUpperCase = name.exists(\_.isUpperCase)

Java代码把字符串看作循环中逐字符步进的低层级实体。Scala代码把同样的字串当作能用论断：predicate查询的字符高层级序列。明显Scala代码更短并且——对训练有素的眼睛来说——比Java代码更容易懂。因此Scala代码在通盘复杂度预算上能极度地变轻，它也更少给你机会犯错 。

论断，\_.isUpperCase，是一个Scala里面函数式文本的例子。它描述了带一个字符参量（用下划线字符代表）的函数，并测试其是否为大写字母 。原则上，这种控制的抽象在Java中也是可能的，为此需要定义一个包含抽象功能的方法的接口。例如，如果你想支持对字串的查询，就应引入一个只有一个方法hasProperty的接口CharacterProperty：

// 在Java里

interface CharacterProperty {

boolean hasProperty(char ch);

}

然后你可以在Java里用这个接口格式实现一个方法exists：它带一个字串和一个CharacterProperty参数，如果字串中有某个字符符合属性，结果返回真。然后你可以这样调用exists：

// 在Java里

exists(name, new CharacterProperty {

boolean hasProperty(char ch) {

return Character.isUpperCase(ch);

}

});

然而，所有这些真的感觉很复杂。复杂到实际上多数Java程序员都不会惹这个麻烦。他们会宁愿写个循环并漠视他们代码里复杂性的累加。另一方面，Scala里的函数式文本真的很轻量，于是就频繁被使用。随着对Scala的逐步了解，你会发现越来越多定义和使用你自己的控制抽象的机会。你将发现这能帮助避免代码重复并因此保持你的程序简短和清晰。

14.3.4 Scala是静态类型的

静态类型系统认定变量和表达式与它们持有和计算的值的种类有关 。Scala坚持作为一种具有非常先进的静态类型系统的语言。从Java那样的内嵌类型系统起步，能够让你使用泛型：generics参数化类型，用交集：intersection联合类型和用抽象类型：abstract type隐藏类型的细节。这些为建造和组织你自己的类型打下了坚实的基础，从而能够设计出即安全又能灵活使用的接口。静态类型系统的经典优越性将更被赏识，其中最重要的包括程序抽象的可检验属性，安全的重构，以及更好的文档。

可检验属性。静态类型系统可以保证消除某些运行时的错误。例如，可以保证这样的属性：布尔型不会与整数型相加；私有变量不会从类的外部被访问；函数带了正确个数的参数；只有字串可以被加到字串集之中 。不过当前的静态类型系统还不能查到其他类型的错误。比方说，通常查不到无法终结的函数，数组越界，或除零错误。同样也查不到你的程序不符合式样书（假设有这么一份式样书）。静态类型系统因此被认为不是很有用而被忽视。有人认为，既然静态类型系统只能发现简单错误，为什么不用能提供更广泛的覆盖的单元测试呢？尽管静态类型系统确实不能替代单元测试，但是却能减少单元测试的数量。同样，单元测试也不能替代静态类型。总而言之，如Edsger Dijkstra所说，测试只能证明存在错误，而非不存在。因此，静态类型能给的保证或许很简单，但它们是无论多少测试都不能给的真正的保证 。

安全的重构。静态类型系统提供了让你具有高度信心改动代码基础的安全网。试想一个对方法加入额外的参数的重构实例。在静态类型语言中，你可以完成修改，重编译你的系统并容易修改所有引起类型错误的代码行。一旦你完成了这些，你确信已经发现了所有需要修改的地方。对其他的简单重构，如改变方法名或把方法从一个类移到另一个，这种确信都有效。所有例子中静态类型检查会提供足够的确认，表明新系统和旧系统可以一样的工作 。

文档。静态类型是被编译器检查过正确性的程序文档。不像普通的注释，类型标注永远都不会过期（至少如果包含它的源文件近期刚刚通过编译就不会）。更进一步说，编译器和集成开发环境可以利用类型标注提供更好的上下文帮助。举例来说，集成开发环境可以通过判定选中表达式的静态类型，找到类型的所有成员，并全部显示出来。

虽然静态类型对程序文档来说通常很有用，当它们弄乱程序时，也会显得很突兀。标准意义上来说，有用的文档是那些程序的读者不可能很容易地从程序中自己想出来的。在如下的方法定义中：

def f(x: String) = ...

知道f的变量应该是String是有用的。另一方面，以下例子中两个标注至少有一个书多余的：

val x: HashMap[Int, String] = new HashMap[Int, String]()

很明显，x是以Int为键，String为值的HashMap这句话说一遍就够了；没必要同样的句子重复两遍。

Scala有非常精于此道的类型推断系统，能让你省略几乎所有的通常被认为是不必要的类型信息。在上例中，以下两个替代语句也能进行同样的工作：

val x = new HashMap[Int, String]()

val x: Map[Int, String] = new HashMap()

Scala里的类型推断可以走的很远。实际上，就算用户代码丝毫没有显式类型也不稀奇。因此，Scala编程经常看上去有点像是动态类型脚本语言写出来的程序。尤其显著表现在作为粘接已写完的库控件的客户应用代码上。而对库控件来说不是这么回事，因为它们常常用到相当精妙的类型去使其适于灵活使用的模式。这很自然。综上，构成可重用控件接口的成员的类型符号应该是显式给出的，因为它们构成了控件和它的使用者间契约的重要部分。

14.4 实验步骤

在spark-shell中编写WordCount代码和运行。

上传in.txt文件到HDFS上。

请大家参照实验一自行完成。

启动spark-shell。

[root@client spark]# cd /usr/cstor/spark

[root@client spark]# bin/spark-shell --master spark://master:7077

写入wordcount的scala代码并运行。

scala> val file=sc.textFile("hdfs://master:8020/user/spark/in/in.txt")

scala> val count=file.flatMap(line => line.split(" ")).map(word => (word,1)).reduceByKey(\_+\_)

scala> count.collect()

14.5 实验结果

14.3.2输入(in.txt) （数据统一放在/root/data/14目录下）

hello world

ni hao

hello my friend

ni are my sunshine

14.3.2结束后运行结果如图14-1所示。

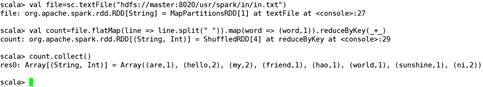


图14-1 运行结果

实验十五 Spark实验：RDD综合实验

15.1 实验目的

1． 通过Spark-shell的操作理解RDD操作；

2． 能通过RDD操作的执行理解RDD的原理；

3． 对Scala能有一定的认识。

15.2 实验要求

在实验结束时能完成max，first，distinct，foreach等api的操作。

15.3 实验原理

RDD（Resilient Distributed Datasets，弹性分布式数据集）是一个分区的只读记录的集合。RDD只能通过在稳定的存储器或其他RDD的数据上的确定性操作来创建。我们把这些操作称作变换以区别其他类型的操作。例如 map、filter和join。

RDD在任何时候都不需要被“物化”（进行实际的变换并最终写入稳定的存储器上）。实际上，一个RDD有足够的信息描述着其如何从其他稳定的存储器上的数据生成。它有一个强大的特性：从本质上说，若RDD失效且不能重建，程序将不能引用该RDD。而用户可以控制RDD的其他两个方面：持久化和分区。用户可以选择重用哪个RDD，并为其制定存储策略(比如：内存存储)。也可以让RDD中的数据根据记录的key分布到集群的多个机器。 这对位置优化来说是有用的，比如可用来保证两个要jion的数据集都使用了相同的哈希分区方式。

编程人员可通过Spark 编程接口对稳定存储上的数据进行变换操作（如map和filter），得到一个或多个RDD。然后可以调用这些RDD的actions（动作）类的操作。这类操作的目是返回一个值或是将数据导入到存储系统中。动作类的操作如count(返回数据集的元素数)，collect（返回元素本身的集合）和save（输出数据集到存储系统）。Spark直到RDD第一次调用一个动作时才真正计算RDD。

还可以调用RDD的persist（持久化）方法来表明该RDD在后续操作中还会用到。默认情况下，Spark会将调用过persist的RDD存在内存中。但若内存不足，也可以将其写入到硬盘上。通过指定persist函数中的参数，用户也可以请求其他持久化策略(如Tachyon)并通过标记来进行persist，比如仅存储到硬盘上或是在各机器之间复制一份。最后，用户可以在每个RDD上设定一个持久化的优先级来指定内存中的哪些数据应该被优先写入到磁盘。 缓存有个缓存管理器，Spark里被称作blockmanager。注意，这里还有一个误区是，很多人认为调用了cache或者persist的那一刻就是在缓存了，这是完全不对的，真正的缓存执行指挥在action被触发。

总结：RDD是分布式只读且已分区集合对象。这些集合是弹性的，如果数据集一部分丢失，则可以对它们进行重建。具有自动容错、位置感知调度和可伸缩性，而容错性是最难实现的，大多数分布式数据集的容错性有两种方式：数据检查点和记录数据的更新。对于大规模数据分析系统，数据检查点操作成本高，主要原因是大规模数据在服务器之间的传输带来的各方面的问题，相比记录数据的更新，RDD也只支持粗粒度的转换，也就是记录如何从其他RDD转换而来（即lineage），以便恢复丢失的分区。

简而言之，特性如下：

1． 数据结构不可变；

2． 支持跨集群的分布式数据操作；

3． 可对数据记录按key进行分区；

4． 提供了粗粒度的转换操作；

5． 数据存储在内存中，保证了低延迟性。

15.4 实验步骤

依据前面实验启动Hadoop和Spark集群。

利用xmanager中的xshell登录到client机器上。

进入到client机器的的Spark的安装目录，执行命令：bin/spark-shell

稍等一段时间，待spark-shell启动之后，屏幕上出现scala的命令提示符之后开始进行Spark命令的键入。

注意：实验需要以本地模式启动：bin/spark-shell，如果以集群模式启动，有可能无法查看输出结果。

15.4.1 distinct 去除RDD内的重复数据

scala > var a = sc.parallelize(List(”Gnu”,”Cat”,”Rat”,”Dog”,”Gnu”,”Rat”),2);

scala > a.distinct.collect

执行输出结果如图15-1所示：

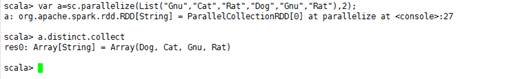


图15-1 去重结果

15.4.2 foreach 遍历RDD内的数据

scala >var b = sc.parallelize(List(”cat”,”dog”,”tiger”,”lion”,”gnu”,”crocodile”,”ant”,”whale”,”dolphin”,”spider”),3)

scala > b.foreach(x=>println(x+”s are yummy”))

执行输出结果如图15-2所示：

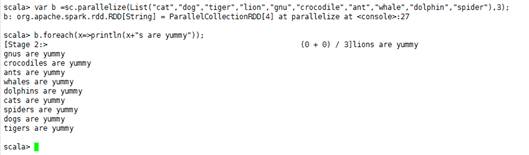


图15-2 遍历结果

15.4.3 first 取的RDD中的第一个数据

scala > var c=sc.parallelize(List(”dog”,”Cat”,”Rat”,”Dog”),2)

scala > c.first

执行输出结果如图15-3所示：

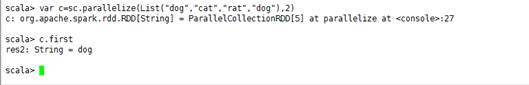


图15-3 first结果

15.4.4 max 取得RDD中的最大的数据

scala > var d=sc.parallelize(10 to 30)

scala > d.max

执行输出结果如图15-4所示：



图15-4 max结果

scala > var e = sc.parallelize(List((10,”dog”),(20,”cat”),(30,”tiger”),(18,”lion”)))

scala > e.max

执行输出结果如图15-5所示：

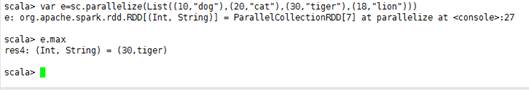


图15-5 max结果

15.4.5 intersection 返回两个RDD重叠的数据

scala > var f = sc.parallelize(1 to 20)

scala > var g = sc.parallelize(10 to 30)

scala > var h = f.intersection(g)

scala > h.collect

执行输出结果如图15-6所示：

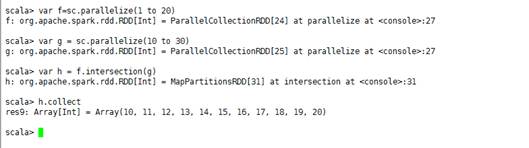


图15-6 重叠结果

实验十六 Spark实验：Spark综例

16.1 实验目的

1． 理解Spark编程思想；

2． 学会在Spark Shell中编写Scala程序；

3． 学会在Spark Shell中运行Scala程序。

16.2 实验要求

实验结束后，能够编写Scala代码解决一下问题，并能够自行分析执行过程。

有三个RDD，要求统计rawRDDA中“aa”、“bb”两个单词出现的次数；要求对去重后的rawRDDA再去掉rawRDDB中的内容；最后将上述两个结果合并成同一个文件然后存入HDFS中。

16.3 实验原理

16.3.1 Scala

Scala是一门多范式的编程语言，一种类似java的编程语言，设计初衷是实现可伸缩的语言、并集成面向对象编程和函数式编程的各种特性。

Scala有几项关键特性表明了它的面向对象的本质。例如，Scala中的每个值都是一个对象，包括基本数据类型（即布尔值、数字等）在内，连函数也是对象。另外，类可以被子类化，而且Scala还提供了基于mixin的组合（mixin-based composition）。

与只支持单继承的语言相比，Scala具有更广泛意义上的类重用。Scala允许定义新类的时候重用“一个类中新增的成员定义（即相较于其父类的差异之处）”。Scala称之为mixin类组合。

Scala还包含了若干函数式语言的关键概念，包括高阶函数（Higher-Order Function）、局部套用（Currying）、嵌套函数（Nested Function）、序列解读（Sequence Comprehensions）等等。

Scala是静态类型的，这就允许它提供泛型类、内部类、甚至多态方法（Polymorphic Method）。另外值得一提的是，Scala被特意设计成能够与Java和.NET互操作。

Scala可以与Java互操作。它用scalac这个编译器把源文件编译成Java的class文件。你可以从Scala中调用所有的Java类库，也同样可以从Java应用程序中调用Scala的代码。

这让Scala得以使用为Java1.4、5.0或者6.0编写的巨量的Java类库和框架，Scala会经常性地针对这几个版本的Java进行测试。Scala可能也可以在更早版本的Java上运行，但没有经过正式的测试。Scala以BSD许可发布，并且数年前就已经被认为相当稳定了。

Scala旨在提供一种编程语言，能够统一和一般化分别来自面向对象和函数式两种不同风格的关键概念。藉着这个目标与设计，Scala得以提供一些出众的特性，包括：

（1）面向对象风格

（2）函数式风格

（3）更高层的并发模型

Scala把Erlang风格的基于actor的并发带进了JVM。开发者可以利用Scala的actor模型在JVM上设计具伸缩性的并发应用程序，它会自动获得多核心处理器带来的优势，而不必依照复杂的Java线程模型来编写程序。

（4）轻量级的函数语法

高阶；

嵌套；

局部套用（Currying）；

匿名。

（5）与XML集成

可在Scala程序中直接书写XML；

可将XML转换成Scala类。

（6）与Java无缝地互操作

总而言之，Scala是一种函数式面向对象语言，它融汇了许多前所未有的特性，而同时又运行于JVM之上。

16.3.2 Spark Shell

该命令用于以交互式方式编写并执行Spark App，且书写语法为Scala。

下面的示例命令用于进入交互式执行器，进入执行器后，即可使用Scala语句以交互式方式编写并执行Spark-App。

[root@client spark]# bin/spark-shell --master spark://master:7077

在该示例中，写明“--master spark://master:7077”的目的是使Spark Shell进入集群模式，若不写明，则Spark Shell会默认进入单机模式。

由于Spark使用Scala开发，而Scala实际上在JVM中执行，因此，我们搭建好Spark环境后，无需另外安装Scala组件。

16.4 实验步骤

16.4.1 启动Spark Shell

登录client服务器，在集群模式下启动Spark Shell，结果如图16-1所示。

[root@client ~]# cd /usr/cstor/spark/

[root@client spark]# bin/spark-shell --master spark://master:7077





图16-1 启动日志

16.4.2 编写并执行Scala代码

在Spark Shell执行器中编写如下scala代码并运行，结果如图16-2所示。

scala> val rawRDDA = sc.parallelize(List("!!bb##cc", "%%ccbb%%", "cc&&++aa"), 3)

scala> val rawRDDB = sc.parallelize(List(("xx", 99), ("yy", 88), ("xx", 99), ("zz", 99)), 2)

scala> val rawRDDC = sc.parallelize(List(("yy",88)), 1)

scala> import org.apache.spark.HashPartitioner

scala> var tempResultRDDA = rawRDDA.flatMap(line=>line.split("")

).filter(allWord=>{allWord.contains("aa") || allWord.contains("bb")}

).map(word=>(word, 1)

).partitionBy(new HashPartitioner(2)

).groupByKey(

).map((P:(String, Iterable[Int]))=>(P.\_1, P.\_2.sum))

scala> var tempResultRDDBC = rawRDDB.distinct.subtract(rawRDDC)

scala> var resultRDDABC = tempResultRDDA.union(tempResultRDDBC)

scala> resultRDDABC.saveAsTextFile("hdfs://master:8020/user/spark/resultRDDABC")

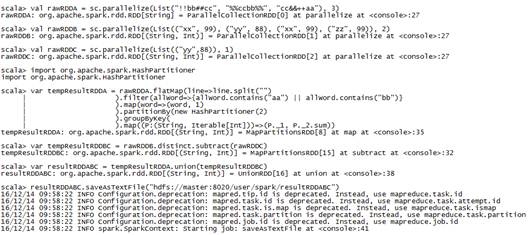


图16-2 代码执行截图

16.4.3 退出Spark Shell

在执行器中执行下列命令，退出Spark Shell，结果如图16-3所示。

scala> exit



图16-3 退出spark-shell

16.4.4 查看执行结果

执行hadoop命令查看运行结果，结果如图16-4所示。

[root@client hadoop]# cd /usr/cstor/hadoop

[root@client hadoop]# bin/hadoop fs -cat /user/spark/resultRDDABC/p\*

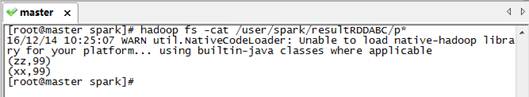


图16-4 执行结果

实验十七 Spark实验：Spark SQL

17.1 实验目的

1． 了解Spark SQL所能实现的功能；

2． 能够使用Spark SQL执行一些sql语句。

17.2 实验要求

1． 能在实验结束之后完成建立数据库，建立数据表的数据结构；

2． 建立数据表之后能在Spark SQL中执行SQL语句进行查询；

3． 练习怎么向Spark SQL中导入数据。

17.3 实验原理

Spark SQL用于以交互式方式编写并执行Spark SQL，且书写语法为类SQL，同Spark Shell一样，启动时写明“--master spark://master:7077”则进入集群模式，否则默认进入单机模式。由于默认安装的Spark已经包含了Spark SQL，故无需安装其它组件，直接执行即可。

Spark SQL使得运行SQL和HiveQL查询十分简单。Spark SQL能够轻易地定位相应的表和元数据。Spark SQL为Spark提供了查询结构化数据的能力，查询时既可以使用SQL也可以使用人们熟知的DataFrame API（RDD）。Spark SQL支持多语言编程包括Java、Scala、Python及R，开发人员可以根据自身喜好进行选择。

DataFrame是Spark SQL的核心，它将数据保存为行构成的集合，行对应列有相应的列名。使用DataFrames可以非常方便地查询数据、给数据绘图及进行数据过滤。

DataFrames也可以用于数据的输入与输出，例如利用Spark SQL中的DataFrames，可以轻易地将下列数据格式加载为表并进行相应的查询操作：

1． RDD；

2． JSON；

3． Hive；

4． Parquet；

5． MySQL；

6． HDFS；

7． S3；

8． JDBC；

9． 其它 。

数据一旦被读取，借助于DataFrames便可以很方便地进行数据过滤、列查询、计数、求平均值及将不同数据源的数据进行整合。

如果你正计划通过读取和写数据来进行分析，Spark SQL可以轻易地帮你实现并将整个过程自动化。

17.4 实验步骤

登录大数据实验一体机，创建Spark集群，并点击搭建Spark集群按钮，等待按钮后方的圆点显示为绿色，即搭建完成。如图17-1所示：



图17-1 平台截图

在master机上建立一个数据文件weather.dat。

[root@master ~]# cat ~/data/16/weather.dat

1 nanjing 16.5

2 shanghai 20.1

3 beijing 12.4

4 zhengzhou 8.3

5 hainan 23.3

6 fujian 24.1

7 hefei 18

[root@master ~]#

在master机上启动Spark SQL。

[root@master ~]# cd /usr/cstor/spark/

[root@master spark]# bin/spark-sql --master spark://master:7077

确认当前Spark SQL中是否已经存在我们需要建立的数据库。

Spark SQL> show databases;

确认在当前的Spark SQL中不存在数据库名为db的数据库时进行操作。

Spark SQL> create database db;

切换当前数据库。

Spark SQL> use db;

建表操作。

Spark SQL> create table weather(

id int,

city string,

temperature double

) row format delimited fields terminated by '\t';

执行命令检查是否建表成功

Spark SQL> show tables;

通过上述命令能在结果中发现weather表。

导入数据

Spark SQL> load data local inpath '/root/data/16/weather.dat' overwrite into table weather;

执行查询命令。

Spark SQL> select \* from weather;

Spark SQL> select \* from weather where temperature > 10.0;

通过查询命令可以正确得到刚才导入的数据就代表导入数据成功。

删除表；

[Spark SQL> drop table weather;

上述命令可以通过查看数据库中存在的表检查weather表是否删除。

17.5 实验结果

建立数据库成功后show databases结果中能看到以下内容，如图17-2所示。

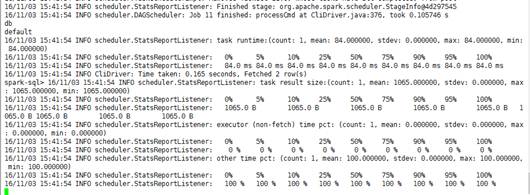


图17-2 show databases结果

建表成功之后show tables 结果能能看到的以下内容，如图17-3所示。

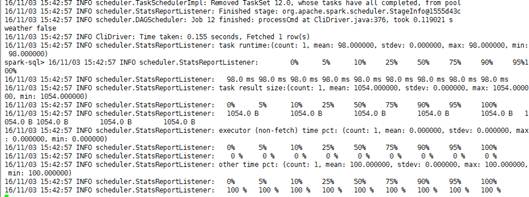


图17-3 show tables结果

执行查询命令成功，如图17-4所示。

select \* from weather;



图17-4 查询结果

select \* from weather where temperature > 10.0，如图17-5所示。

https://bd.cstor.cn/experiment/exp17.files/image005.jpg

图17-5 查询结果

实验十八 Spark实验：Spark Streaming

18.1 实验目的

1． 了解Spark Streaming版本的WordCount和MapReduce版本的WordCount的区别；

2． 理解Spark Streaming的工作流程；

3． 理解Spark Streaming的工作原理。

18.2 实验要求

要求实验结束时，每位学生能正确运行成功本实验中所写的jar包程序，能正确的计算出单词数目。

18.3 实验原理

18.3.1 Spark Streaming架构

计算流程：Spark Streaming是将流式计算分解成一系列短小的批处理作业。这里的批处理引擎是Spark，也就是把Spark Streaming的输入数据按照batch size（如1秒）分成一段一段的数据（Discretized Stream），每一段数据都转换成Spark中的RDD（Resilient Distributed Dataset），然后将Spark Streaming中对DStream的Transformation操作变为针对Spark中对RDD的Transformation操作，将RDD经过操作变成中间结果保存在内存中。整个流式计算根据业务的需求可以对中间的结果进行叠加，或者存储到外部设备。如图18-1所示：

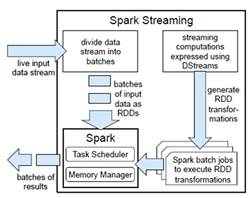


图18-1 spark-streaming计算流程

容错性：对于流式计算来说，容错性至关重要。首先我们要明确一下Spark中RDD的容错机制。每一个RDD都是一个不可变的分布式可重算的数据集，其记录着确定性的操作继承关系（lineage），所以只要输入数据是可容错的，那么任意一个RDD的分区（Partition）出错或不可用，都是可以利用原始输入数据通过转换操作而重新算出的。

对于Spark Streaming来说，其RDD的传承关系如下图所示，图中的每一个椭圆形表示一个RDD，椭圆形中的每个圆形代表一个RDD中的一个Partition，图中的每一列的多个RDD表示一个DStream（图中有三个DStream），而每一行最后一个RDD则表示每一个Batch Size所产生的中间结果RDD。我们可以看到图中的每一个RDD都是通过lineage相连接的，由于Spark Streaming输入数据可以来自于磁盘，例如HDFS（多份拷贝）或是来自于网络的数据流（Spark Streaming会将网络输入数据的每一个数据流拷贝两份到其他的机器）都能保证容错性。所以RDD中任意的Partition出错，都可以并行地在其他机器上将缺失的Partition计算出来。这个容错恢复方式比连续计算模型（如Storm）的效率更高。 如图18-2所示：

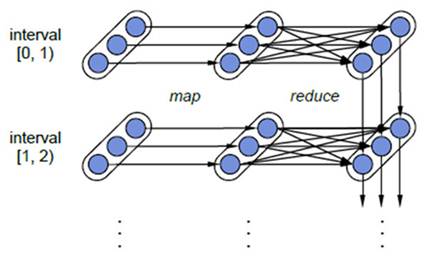


图18-2 rdd关系

实时性：对于实时性的讨论，会牵涉到流式处理框架的应用场景。Spark Streaming将流式计算分解成多个Spark Job，对于每一段数据的处理都会经过Spark DAG图分解，以及Spark的任务集的调度过程。对于目前版本的Spark Streaming而言，其最小的Batch Size的选取在0.5~2秒钟之间（Storm目前最小的延迟是100ms左右），所以Spark Streaming能够满足除对实时性要求非常高（如高频实时交易）之外的所有流式准实时计算场景。

扩展性与吞吐量：Spark目前在EC2上已能够线性扩展到100个节点（每个节点4Core），可以以数秒的延迟处理6GB/s的数据量（60M records/s），其吞吐量也比流行的Storm高2～5倍，图4是Berkeley利用WordCount和Grep两个用例所做的测试，在Grep这个测试中，Spark Streaming中的每个节点的吞吐量是670k records/s，而Storm是115k records/s。如图18-3所示：

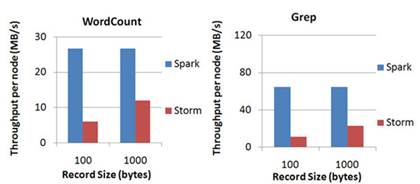


图18-3 性能比较

18.3.2 Spark Streaming编程模型

Spark Streaming的编程和Spark的编程如出一辙，对于编程的理解也非常类似。对于Spark来说，编程就是对于RDD的操作；而对于Spark Streaming来说，就是对DStream的操作。下面将通过一个大家熟悉的WordCount的例子来说明Spark Streaming中的输入操作、转换操作和输出操作。

Spark Streaming初始化：在开始进行DStream操作之前，需要对Spark Streaming进行初始化生成StreamingContext。参数中比较重要的是第一个和第三个，第一个参数是指定Spark Streaming运行的集群地址，而第三个参数是指定Spark Streaming运行时的batch窗口大小。在这个例子中就是将1秒钟的输入数据进行一次Spark Job处理。

val ssc = new StreamingContext("Spark://…", "WordCount", Seconds(1), [Homes], [Jars])

Spark Streaming的输入操作：目前Spark Streaming已支持了丰富的输入接口，大致分为两类：一类是磁盘输入，如以batch size作为时间间隔监控HDFS文件系统的某个目录，将目录中内容的变化作为Spark Streaming的输入；另一类就是网络流的方式，目前支持Kafka、Flume、Twitter和TCP socket。在WordCount例子中，假定通过网络socket作为输入流，监听某个特定的端口，最后得出输入DStream（lines）。

val lines = ssc.socketTextStream("localhost",8888)

Spark Streaming的转换操作：与Spark RDD的操作极为类似，Spark Streaming也就是通过转换操作将一个或多个DStream转换成新的DStream。常用的操作包括map、filter、flatmap和join，以及需要进行shuffle操作的groupByKey/reduceByKey等。在WordCount例子中，我们首先需要将DStream(lines)切分成单词，然后将相同单词的数量进行叠加, 最终得到的wordCounts就是每一个batch size的（单词，数量）中间结果。

val words = lines.flatMap(\_.split(" "))

val wordCounts = words.map(x => (x, 1)).reduceByKey(\_ + \_)

另外，Spark Streaming有特定的窗口操作，窗口操作涉及两个参数：一个是滑动窗口的宽度（Window Duration）；另一个是窗口滑动的频率（Slide Duration），这两个参数必须是batch size的倍数。例如以过去5秒钟为一个输入窗口，每1秒统计一下WordCount，那么我们会将过去5秒钟的每一秒钟的WordCount都进行统计，然后进行叠加，得出这个窗口中的单词统计。

val wordCounts = words.map(x => (x, 1)).reduceByKeyAndWindow(\_ + \_, Seconds(5s)，seconds(1))

但上面这种方式还不够高效。如果我们以增量的方式来计算就更加高效，例如，计算t+4秒这个时刻过去5秒窗口的WordCount，那么我们可以将t+3时刻过去5秒的统计量加上[t+3，t+4]的统计量，在减去[t-2，t-1]的统计量，这种方法可以复用中间三秒的统计量，提高统计的效率。如图18-4所示：

val wordCounts = words.map(x => (x, 1)).reduceByKeyAndWindow(\_ + \_, \_ - \_, Seconds(5s)，seconds(1))



图18-4 执行过程

Spark Streaming的输出操作：对于输出操作，Spark提供了将数据打印到屏幕及输入到文件中。在WordCount中我们将DStream wordCounts输入到HDFS文件中。

wordCounts = saveAsHadoopFiles("WordCount")

Spark Streaming启动：经过上述的操作，Spark Streaming还没有进行工作，我们还需要调用Start操作，Spark Streaming才开始监听相应的端口，然后收取数据，并进行统计。

ssc.start()

18.3.3 Spark Streaming典型案例

在互联网应用中，网站流量统计作为一种常用的应用模式，需要在不同粒度上对不同数据进行统计，既有实时性的需求，又需要涉及到聚合、去重、连接等较为复杂的统计需求。传统上，若是使用Hadoop MapReduce框架，虽然可以容易地实现较为复杂的统计需求，但实时性却无法得到保证；反之若是采用Storm这样的流式框架，实时性虽可以得到保证，但需求的实现复杂度也大大提高了。Spark Streaming在两者之间找到了一个平衡点，能够以准实时的方式容易地实现较为复杂的统计需求。 下面介绍一下使用Kafka和Spark Streaming搭建实时流量统计框架。

数据暂存：Kafka作为分布式消息队列，既有非常优秀的吞吐量，又有较高的可靠性和扩展性，在这里采用Kafka作为日志传递中间件来接收日志，抓取客户端发送的流量日志，同时接受Spark Streaming的请求，将流量日志按序发送给Spark Streaming集群。

数据处理：将Spark Streaming集群与Kafka集群对接，Spark Streaming从Kafka集群中获取流量日志并进行处理。Spark Streaming会实时地从Kafka集群中获取数据并将其存储在内部的可用内存空间中。当每一个batch窗口到来时，便对这些数据进行处理。

结果存储：为了便于前端展示和页面请求，处理得到的结果将写入到数据库中。

相比于传统的处理框架，Kafka+Spark Streaming的架构有以下几个优点。Spark框架的高效和低延迟保证了Spark Streaming操作的准实时性。利用Spark框架提供的丰富API和高灵活性，可以精简地写出较为复杂的算法。编程模型的高度一致使得上手Spark Streaming相当容易，同时也可以保证业务逻辑在实时处理和批处理上的复用。

Spark Streaming提供了一套高效、可容错的准实时大规模流式处理框架，它能和批处理及即时查询放在同一个软件栈中。如果你学会了Spark编程，那么也就学会了Spark Streaming编程，如果理解了Spark的调度和存储，Spark Streaming也类似。按照目前的发展趋势，Spark Streaming一定将会得到更大范围的使用。

18.4 实验步骤

登录大数据实验一体机，创建Spark集群，并点击搭建Spark集群按钮，等待按钮后方的圆点显示为绿色，即搭建完成。如图18-5所示：



图18-5 平台截图

使用jps检验Hadoop集群和Spark集群是否成功启动。成功启动Hadoop集群和Spark集群的的情况使用jps命令能成功看到以下java进程。

[root@master ~]# jps

3711 NameNode

4174 ResourceManager

3957 SecondaryNameNode

4738 Jps

4635 Master

打开IntelliJ IDEA 准备编写Spark-steaming代码。

点击File -> New -> Module-> Maven->Next –> 输入GroupId和AriifactId -> Next -> 输入Module name 新建一个maven的Module。

打开项目录，点击目录下的pom.xml文件，在<project>标签中输入maven的依赖。然后右键->maven->Reimport导入maven依赖， 效果如下：

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.cstor.sparkstreaming</groupId>

<artifactId>nice</artifactId>

<version>1.0-SNAPSHOT</version>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<source>1.6</source>

<target>1.6</target>

</configuration>

</plugin>

</plugins>

</build>

<!-- https://mvnrepository.com/artifact/org.apache.spark/Spark Streaming\_2.10 -->

<dependencies>

<dependency>

<groupId>org.apache.spark</groupId>

<artifactId>spark-streaming\_2.10</artifactId>

<version>1.6.0</version>

</dependency>

</dependencies>

</project>

在src/main/java的目录下，点击java目录新建一个package命名为spark.streaming.test，然后在包下新建一个SparkStreaming的java class。

在SparkStreaming中键入代码。

package spark.streaming.test;

import scala.Tuple2;

import com.google.common.collect.Lists;

import org.apache.spark.SparkConf;

import org.apache.spark.api.java.function.FlatMapFunction;

import org.apache.spark.api.java.function.Function2;

import org.apache.spark.api.java.function.PairFunction;

import org.apache.spark.api.java.StorageLevels;

import org.apache.spark.streaming.Durations;

import org.apache.spark.streaming.api.java.JavaDStream;

import org.apache.spark.streaming.api.java.JavaPairDStream;

import org.apache.spark.streaming.api.java.JavaReceiverInputDStream;

import org.apache.spark.streaming.api.java.JavaStreamingContext;

import java.util.Iterator;

import java.util.regex.Pattern;

public class SparkStreaming {

private static final Pattern SPACE = Pattern.compile(" ");

public static void main(String[] args) throws InterruptedException {

if (args.length < 2) {

System.err.println("Usage: JavaNetworkWordCount <hostname> <port>");

System.exit(1);

}

SparkConf sparkConf = new SparkConf().setAppName("JavaNetworkWordCount");

JavaStreamingContext ssc = new JavaStreamingContext(sparkConf, Durations.seconds(1));

JavaReceiverInputDStream<String> lines = ssc.socketTextStream(

args[0], Integer.parseInt(args[1]), StorageLevels.MEMORY\_AND\_DISK\_SER);

JavaDStream<String> words = lines.flatMap(new FlatMapFunction<String, String>() {

@Override

public Iterable<String> call(String x){

return Lists.newArrayList(SPACE.split(x));

}

});

JavaPairDStream<String, Integer> wordCounts = words.mapToPair(

new PairFunction<String, String, Integer>() {

@Override

public Tuple2<String, Integer> call(String s) {

return new Tuple2<String, Integer>(s, 1);

}

}).reduceByKey(new Function2<Integer, Integer, Integer>() {

@Override

public Integer call(Integer i1, Integer i2) {

return i1 + i2;

}

});

wordCounts.print();

ssc.start();

ssc.awaitTermination();

}

}

点击File -> Project Structure -> Aritifacts –>点击加号 -> JAR -> from modules with dependences -> 选择刚才新建的module –> 选择Main Class -> Ok -> 选择Output directory –>点击Ok。

去掉除 ’guava-14.0.1.jar’ 和 ‘guice-3.0.jar’以外所有的JAR 包，点击Ok。

点击Build -> Build Aritifacts 。选择刚才设置的jar包，上传到master上去。

新建一个SSH连接，登录master服务器，使用命令nc -lk 9999设置路由器。

[root@master ~]# nc -lk 9999

注：如果系统中没有nc这个命令，可以使用yum install nc安装nc命令。

进入spark的安装目录，执行下面的命令。

[root@master ~]# cd /usr/cstor/spark

[root@master spark]# bin/spark-submit --class spark.streaming.test.SparkStreaming ~/sparkstreaming.jar localhost 9999

在网络流中输入单词。按回车结束一次输出。

在命令提交的xshell连接中观察程序输出。

18.5 实验结果

在提交任务之后应该能看到以下结果（因屏幕刷新很快，所以只能看到部分结果）。

在nc -lk 9999 命令下输入，如图18-6所示。

https://bd.cstor.cn/experiment/exp18.files/image006.jpg

图18-6 执行结果

所示结果中应该立刻显示出如下内容，如图18-7所示。

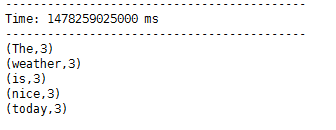


图18-7 执行结果

实验十九 Spark实验：GraphX

19.1 实验目的

1． 了解Spark的图计算框架GraphX的基本知识；

2． 能利用GraphX进行建图；

3． 能利用GraphX进行基本的图操作；

4． 理解GraphX图操作的算法。

19.2 实验要求

要求实验结束时，每位学生能完成正确运行Spark GraphX的示例程序，正确上传到集群中运行得到正确的实验结果。

实验结束时能对实验代码进行一定的理解。

19.3 实验原理

Spark GraphX是一个分布式图处理框架，Spark GraphX基于Spark平台提供对图计算和图挖掘简洁易用的而丰富多彩的接口，极大的方便了大家对分布式图处理的需求。

社交网络中人与人之间有很多关系链，例如Twitter、Facebook、微博、微信，这些都是大数据产生的地方，都需要图计算，现在的图处理基本都是分布式的图处理，而并非单机处理，Spark GraphX由于底层是基于Spark来处理的，所以天然就是一个分布式的图处理系统。

图的分布式或者并行处理其实是把这张图拆分成很多的子图，然后我们分别对这些子图进行计算，计算的时候可以分别迭代进行分阶段的计算，即对图进行并行计算。

适用范围：图计算。

19.4 实验步骤

本实验主要可以分为在IDEA安装Scala的插件、编写Scala的程序、打包提交程序和查看程序运行结果等几个步骤。详细步骤如下：

确认好上述“操作前提”后，读者可按下述步骤执行

19.4.1 在Intellij IDEA 中安装Scala的插件

注：IntelliJ IDEA对Scala编程比较方便。如不习惯请参考网上Eclipse版本。

点击File -> Plugins -> 输入”Scala” 搜索 -> 点击搜索条目为Scala的项目-> 点击右侧绿色按钮Install -> 安装完成之后重启IDEA。

19.4.2 新建Scala Module

点击当前IDEA中的project然后右键点击new -> module -> 选择Scala -> next -> 输入module name “sparkgraphx”。

点击新建的sparkgraphx 的module 然后右键点击”A dd Framework Support” 选择maven支持。

19.4.3 添加maven依赖

在pom.xml 文件中添加如下依赖

<dependencies>

<dependency>

<groupId>org.apache.spark</groupId>

<artifactId>spark-graphx\_2.10</artifactId>

<version>1.5.1</version>

</dependency>

</dependencies>

右键选项点击Reimport导入依赖。（需要稍等一段时间导入maven依赖）。

19.4.4 新建Scala程序

打开项目目录然后右键新建package命名为test，然后新建Scala class 将Kindt由class改为object。输入名字为GraphXExample。

输入代码：

package test

import org.apache.log4j.{Level, Logger}

import org.apache.spark.graphx.\_

import org.apache.spark.rdd.RDD

import org.apache.spark.{SparkConf, SparkContext}

object GraphXExample {

def main(args: Array[String]) {

Logger.getLogger("org.apache.spark").setLevel(Level.ERROR)

Logger.getLogger("org.eclipse.jetty.server").setLevel(Level.OFF)

val conf = new SparkConf().setAppName("SimpleGraphX").setMaster("local")

val sc = new SparkContext(conf)

val vertexArray = Array(

(1L, ("Alice", 28)),

(2L, ("Bob", 27)),

(3L, ("Charlie", 65)),

(4L, ("David", 42)),

(5L, ("Ed", 55)),

(6L, ("Fran", 50))

)

val edgeArray = Array(

Edge(2L, 1L, 7),

Edge(2L, 4L, 2),

Edge(3L, 2L, 4),

Edge(3L, 6L, 3),

Edge(4L, 1L, 1),

Edge(5L, 2L, 2),

Edge(5L, 3L, 8),

Edge(5L, 6L, 3)

)

val vertexRDD: RDD[(Long, (String, Int))] = sc.parallelize(vertexArray)

val edgeRDD: RDD[Edge[Int]] = sc.parallelize(edgeArray)

val graph: Graph[(String, Int), Int] = Graph(vertexRDD, edgeRDD)

println("属性演示")

println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

println("找出图中年龄大于30的顶点：")

graph.vertices.filter { case (id, (name, age)) => age > 30}.collect.foreach {

case (id, (name, age)) => println(s"$name is $age")

}

println("找出图中属性大于5的边：")

graph.edges.filter(e => e.attr > 5).collect.foreach(e => println(s"${e.srcId} to ${e.dstId} att ${e.attr}"))

println

println("列出边属性>5的tripltes：")

for (triplet <- graph.triplets.filter(t => t.attr > 5).collect) {

println(s"${triplet.srcAttr.\_1} likes ${triplet.dstAttr.\_1}")

}

println

println("找出图中最大的出度、入度、度数：")

def max(a: (VertexId, Int), b: (VertexId, Int)): (VertexId, Int) = {

if (a.\_2 > b.\_2) a else b

}

println("max of outDegrees:" + graph.outDegrees.reduce(max) + " max of inDegrees:" + graph.inDegrees.reduce(max) + " max of Degrees:" + graph.degrees.reduce(max))

println

println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

println("转换操作")

println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

println("顶点的转换操作，顶点age + 10：")

graph.mapVertices{ case (id, (name, age)) => (id, (name, age+10))}.vertices.collect.foreach(v => println(s"${v.\_2.\_1} is ${v.\_2.\_2}"))

println

println("边的转换操作，边的属性\*2：")

graph.mapEdges(e=>e.attr\*2).edges.collect.foreach(e => println(s"${e.srcId} to ${e.dstId} att ${e.attr}"))

println

println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

println("结构操作")

println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

println("顶点年纪>30的子图：")

val subGraph = graph.subgraph(vpred = (id, vd) => vd.\_2 >= 30)

println("子图所有顶点：")

subGraph.vertices.collect.foreach(v => println(s"${v.\_2.\_1} is ${v.\_2.\_2}"))

println

println("子图所有边：")

subGraph.edges.collect.foreach(e => println(s"${e.srcId} to ${e.dstId} att ${e.attr}"))

println

println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

println("连接操作")

println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

val inDegrees: VertexRDD[Int] = graph.inDegrees

case class User(name: String, age: Int, inDeg: Int, outDeg: Int)

val initialUserGraph: Graph[User, Int] = graph.mapVertices { case (id, (name, age)) => User(name, age, 0, 0)}

val userGraph = initialUserGraph.outerJoinVertices(initialUserGraph.inDegrees) {

case (id, u, inDegOpt) => User(u.name, u.age, inDegOpt.getOrElse(0), u.outDeg)

}.outerJoinVertices(initialUserGraph.outDegrees) {

case (id, u, outDegOpt) => User(u.name, u.age, u.inDeg,outDegOpt.getOrElse(0))

}

println("连接图的属性：")

userGraph.vertices.collect.foreach(v => println(s"${v.\_2.name} inDeg: ${v.\_2.inDeg} outDeg: ${v.\_2.outDeg}"))

println

println("出度和入度相同的人员：")

userGraph.vertices.filter {

case (id, u) => u.inDeg == u.outDeg

}.collect.foreach {

case (id, property) => println(property.name)

}

println

println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

println("聚合操作")

println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

println("找出5到各顶点的最短：")

val sourceId: VertexId = 5L

val initialGraph = graph.mapVertices((id, \_) => if (id == sourceId) 0.0 else Double.PositiveInfinity)

val sssp = initialGraph.pregel(Double.PositiveInfinity)(

(id, dist, newDist) => math.min(dist, newDist),

triplet => {

if (triplet.srcAttr + triplet.attr < triplet.dstAttr) {

Iterator((triplet.dstId, triplet.srcAttr + triplet.attr))

} else {

Iterator.empty

}

},

(a,b) => math.min(a,b)

)

println(sssp.vertices.collect.mkString("\n"))

sc.stop()

}

}

19.4.5 程序运行

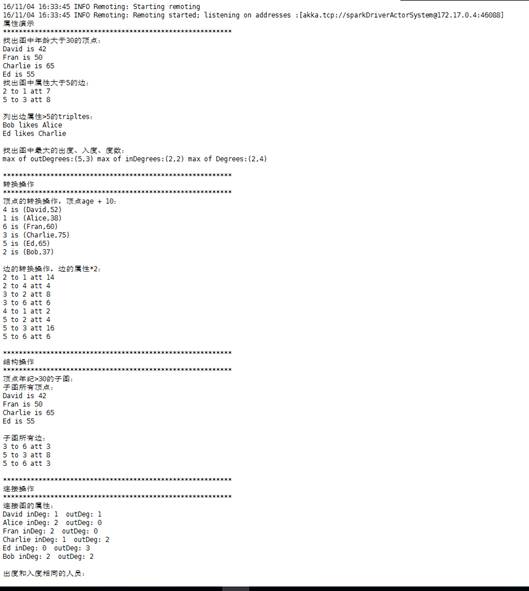
输入命令运行：

[root@client ~]# cd /usr/cstor /spark/

[root@client spark-1.6.0/]# bin/spark-submit --class test.GraphXExample ~/ sparkgraphx.jar

19.5 实验结果

程序正确运行结果如图19-1所示：



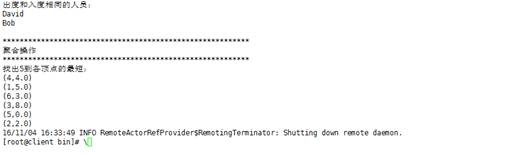


图19-1 执行结果

实验二十 部署ZooKeeper

20.1 实验目的

掌握ZooKeeper集群安装部署，加深对ZooKeeper相关概念的理解，熟练ZooKeeper的一些常用Shell命令。

20.2 实验要求

部署三个节点的ZooKeeper集群，通过ZooKeeper客户端连接ZooKeeper集群，并用Shell命令练习创建目录，查询目录等。

20.3 实验原理

ZooKeeper 分布式服务框架是 Apache Hadoop 的一个子项目，它主要是用来解决分布式应用中经常遇到的一些数据管理问题，如：统一命名服务、状态同步服务、集群管理、分布式应用配置项的管理等。

ZooKeeper是以Fast Paxos算法为基础的。

ZooKeeper集群的初始化过程：集群中所有机器以投票的方式（少数服从多数）选取某一台机器作为leader(领导者)，其余机器作为follower(追随者)。如果集群中只有一台机器，那么就这台机器就是leader，没有follower。

ZooKeeper集群与客户端的交互：客户端可以在任意情况下ZooKeeper集群中任意一台机器上进行读操作；但是写操作必须得到leader的同意后才能执行。

ZooKeeper选取leader的核心算法思想：如果某服务器获得N/2 + 1票，则该服务器成为leader。N为集群中机器数量。为了避免出现两台服务器获得相同票数（N/2），应该确保N为奇数。因此构建ZooKeeper集群最少需要3台机器。

20.4 实验步骤

本实验主要介绍ZooKeeper的部署，ZooKeeper一般部署奇数个节点，部署方法包主要含安装JDK、修改配置文件、启动测试三个步骤。

20.4.1 安装JDK

下载安装JDK。因为 ZooKeeper 服务器在 JVM 上运行。

20.4.2 修改ZooKeeper配置文件

首先配置slave1，slave2,slave3之间的免密和各个机器的/etc/hosts文件

修改ZooKeeper的配置文件，步骤如下：

进入解压目录下，把conf目录下的zoo\_sample.cfg 复制成zoo.cfg文件。

cd /usr/cstor/zookeeper/conf

cp zoo\_sample.cfg zoo.cfg

打开zoo.cfg并修改和添加配置项目，如下：

# The number of milliseconds of each tick  
 tickTime=2000  
 # The number of ticks that the initial   
 # synchronization phase can take  
 initLimit=10  
 # The number of ticks that can pass between   
 # sending a request and getting an acknowledgement  
 syncLimit=5  
 # the port at which the clients will connect  
 clientPort=2181  
 # the directory where the snapshot is stored.  
 dataDir=/usr/cstor/zookeeper/data

dataLogDir=/usr/cstor/zookeeper/log

server.1=slave1:2888:3888

server.2=slave2:2888:3888

server.3=slave3:2888:3888

新建两个目录。

mkdir /usr/cstor/zookeeper/data

mkdir /usr/cstor/zookeeper/log

将/usr/cstor/zookeeper目录传到另外两台机器上。

scp -r /usr/cstor/zookeeper root@slave2:/usr/cstor

scp -r /usr/cstor/zookeeper root@slave3:/usr/cstor

分别在三个节点上的/usr/local/zookeeper/data目录下创建一个文件：myid。

vi /usr/cstor/zookeeper/data/myid

分别在myid上按照配置文件的server.<id>中id的数值，在不同机器上的该文 件中填写相应过的值，如下：

slave1 的myid内容为1

slave2 的myid内容为2  
 slave3 的myid内容为3

20.4.3 启动ZooKeeper集群

然后，启动ZooKeeper集群，进入客户端验证部署完成。

分别在三个节点进入bin目录，启动ZooKeeper服务进程：

cd /usr/cstor/zookeeper/bin

./zkServer.sh start

在各机器上依次执行脚本，查看ZooKeeper状态信息，两个节点是follower状态，一个节点是leader状态：

./zkServer.sh status

在其中一台机器上执行客户端脚本：

./zkCli.sh -server slave1:2181,slave2:2181,slave3:2181

在客户端shell下执行创建目录命令：

create /testZk ""

向/testZk目录写数据：

set /testZk 'aaa'

读取/testZk目录数据：

get /testZk

删除/testZk目录：

rmr /testZk

在客户端shell下用quit命令退出客户端：

quit

20.5 实验结果

各个节点执行jps命令查看Java进程，有QuorumPeerMain进程代表该节点ZooKeeper安装成功，如图20-1所示。

https://bd.cstor.cn/experiment/exp20.files/image001.png

图20-1 ZooKeeper进程图

在客户端shell下查看ZooKeeper集群目录,输入命令：ls /，查看ZooKeeper集群目录列表结果如图20-2、图20-3、图20-4所示。



图20-2 ZooKeeper客户端结果图



图20-3 ZooKeeper客户端写数据

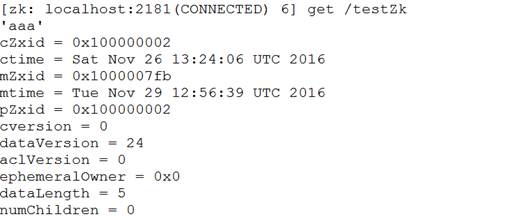


图20-4 ZooKeeper客户端读数据

实验二十一 ZooKeeper进程协作

21.1 实验目的

掌握Java代码如何连接ZooKeeper集群及通过代码读写ZooKeeper集群的目录下的数据，掌握ZooKeeper如何实现多个线程间的协作。

21.2 实验要求

用Java代码实现两个线程，一个向ZooKeeper中某一目录中写入数据，另一线程监听此目录，若目录下数据有更新则将目录中数据读取并显示出来。

21.3 实验原理

通过ZooKeeper实现不同物理机器上的进程间通信。

场景使用：客户端A需要向客户端B发送一条消息msg1。

实现方法：客户端A把msg1发送给ZooKeeper集群，然后由客户端B

自行去ZooKeeper集群读取msg1。

21.4 实验步骤

本实验主要完成多线程通过ZooKeeper完成彼此间的协作问题，实验过程包含启动集群、编写代码、客户端提交代码三个步骤。

21.4.1 启动ZooKeeper集群

启动ZooKeeper集群。具体步骤可以参考实验20。

21.4.2 导入jar包

其次，从ZooKeeper安装包的lib目录下，将如下jar包导入到开发工具：

jline-0.9.94.jar  
 log4j-1.2.16.jar  
 netty-3.7.0.Final.jar  
 slf4j-api-1.6.1.jar  
 slf4j-log4j12-1.6.1.jar  
 zookeeper-3.4.6.jar

21.4.3 编写java代码

在开发工具编写Java代码，完成实验要求的功能，代码如下：

向/testZk目录写数据线程代码实现：

public class WriteMsg extends Thread {

@Override

public void run() {

try {

ZooKeeper zk = new ZooKeeper("slave1:2181", 500000, null);

String content = Long.toString(new Date().getTime());

// 修改节点/testZk下的数据，第三个参数为版本，如果是-1，那会无视被

修改的数据版本，直接改掉

zk.setData("/testZk", content.getBytes(), -1);

// 关闭session

zk.close();

} catch (Exception e) {

e.printStackTrace();

}

}

}

监听/testZk目录若数据改变则读取数据并显示线程代码实现：

public class ReadMsg {

public static void main(String[] args) throws Exception {

final ZooKeeper zk = new ZooKeeper("slave1:2181", 500000, null);

//定义watch

Watcher wacher = new Watcher() {

public void process(WatchedEvent event) {

//监听到数据变化取出数据

if(EventType.NodeDataChanged == event.getType()){

byte[] bb;

try {

bb = zk.getData("/testZk", null, null);

System.out.println("/testZk的数据: "+new String(bb));

} catch (Exception e) {

e.printStackTrace();

}

}

}

};

//设置watch

zk.exists("/testZk", wacher);

//更新/testZk目录信息，触发wacth

while(true)

{

Thread.sleep(2000);

new WriteMsg().start();

//watch一次生效就会删除需重新设置

zk.exists("/testZk", wacher);

}

}

}

21.4.4 做成jar包

将上述代码打成jar包，用工具上传到客户端节点，并执行代码：

java -jar ZooKeeperTest.jar

21.5 实验结果

在客户端提交jar包后运行Java代码，ZooKeeper接收线程监控/testZk目录信息是否有变化，有变化时，读取该目录的内容并打印到日志中，结果如图21-1所示。

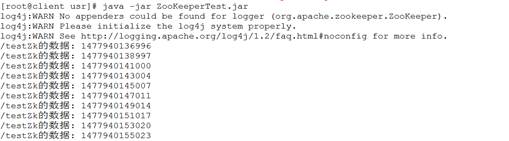


图21-1 Java代码运行结果图

实验二十二 部署HBase

22.1 实验目的

1． 掌握HBase基础简介及体系架构；

2． 掌握HBase集群安装部署及HBase Shell的一些常用命令的使用；

3． 了解HBase和HDFS及Zookeeper之间的关系。

22.2 实验要求

1． 巩固学习下实验一、实验二、实验二十；

2． 部署一个主节点，三个子节点的HBase集群，并引用外部Zookeeper；

3． 进入HBase Shell通过命令练习创建表、插入数据及查询等命令。

22.3 实验原理

简介：HBase是基于Hadoop的开源分布式数据库，它以Google的BigTable为原型，设计并实现了具有高可靠性、高性能、列存储、可伸缩、实时读写的分布式数据库系统，它是基于列而不是基于行的模式，适合存储非结构化数据。

体系结构：HBase是一个分布式的数据库，使用Zookeeper管理集群，使用HDFS作为底层存储，它由HMaster和HRegionServer组成，遵从主从服务器架构。HBase将逻辑上的表划分成多个数据块即HRegion，存储在HRegionServer中。HMaster负责管理所有的HRegionServer，它本身并不存储任何数据，而只是存储数据到HRegionServer的映射关系(元数据)。HBase的基本架构如图22-1所示：

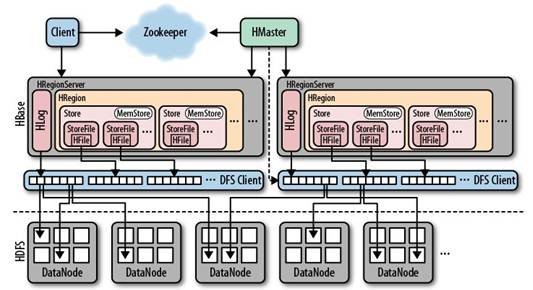


图22-1 HBase的基本架构

22.4 实验步骤

本实验主要演示HBase的安装部署过程，因HBase依赖于HDFS和Zookeeper，所以该实验需要分为四个步骤。

首先，配置SSH无密钥登录(参考实验1)。

其次，安装Hadoop集群(参考实验2)。

然后，安装Zookeeper集群(参考实验20)。

最后，修改HBase配置文件，具体内容如下：

将HBase 安装包hbase.1.1.2.tar.gz解压到/usr/cstor目录，并将hbase.1.1.2目录改名为hbase，且所属用户改成root:root。

[root@master ~]#tar -zxvf hbase.1.1.2.tar.gz -c /usr/cstor/hbase

[root@master ~]#mv /usr/cstor/hbase.1.1.2 /usr/cstor/hbase

[root@master ~]#chown -R root:root /usr/cstor/hbase

进入解压目录下，配置conf目录下的/usr/cstor/hbase/conf/hbase-env.sh文件，设置如下：

#Java安装路径

export JAVA\_HOME=/usr/local/jdk1.7.0\_79 (需根据实际情况指定)

#不使用HBase自带的Zookeeper

export HBASE\_MANAGES\_ZK=false

配置conf目录下的hbase-site.xml文件，设置如下：

<configuration>

<property>

<name>hbase.rootdir</name>

<value>hdfs://master:8020/hbase</value>

</property>

<property>

<name>hbase.cluster.distributed</name>

<value>true</value>

</property>

<property>

<name>hbase.zookeeper.quorum</name>

<value>slave1,slave2,slave3</value>

</property>

<property>

<name>hbase.tmp.dir</name>

<value>/usr/cstor/hbase/data/tmp</value>

</property>

</configuration>

配置conf目录下的regionservers文件，设置如下：

slave1

slave2

slave3

配置完成后，将hbase目录传输到集群的其它节点。

scp -r /usr/cstor/hbase root@slave1:/usr/cstor

scp -r /usr/cstor/hbase root@slave2:/usr/cstor

scp -r /usr/cstor/hbase root@slave3:/usr/cstor

接着，启动HBase，并简单验证HBase，如下：

在主节点master进入hbase解压目录的bin目录，启动HBase服务进程(已启动Zookeeper)：

[root@master ~]#cd /usr/cstor/hbase/bin

./start-hbase.sh

启动信息如图22-2所示：



图22-2 启动信息

通过以下命令进入HBase shell界面：

./hbase shell

在shell里创建表：

create 'testhbase' , 'f1'

查询所有表名：

list

查看表结构信息：

describe 'testhbase'

在shell里插入数据：

put 'testhbase', '001', 'f1:name', 'aaa'

在shell里查询：

scan 'testhbase'

删除表，先disable再drop：

disable 'testhbase'

drop 'testhbase'

22.5 实验结果

HBase启动成功后，进入shell界面，用shell命令简单操作HBase数据库验证HBase成功安装，验证结果如图22-3所示。

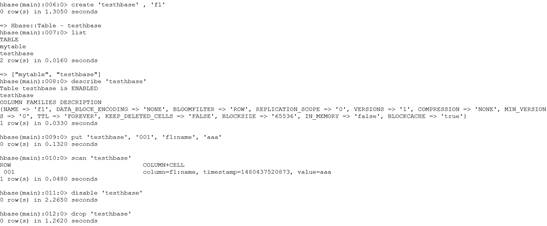


图22-3 HBase Shell界面图

HBase安装成功后，可以通过访问HBase WEB页面（http://master:16010）来查看HBase集群的一些基本情况，如图22-4所示。

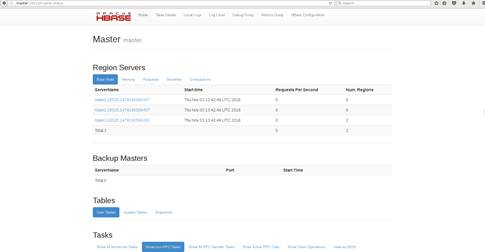


图22-4 HBase管理界面

实验二十三 新建HBase表

23.1 实验目的

1． 掌握HBase数据模型(逻辑模型及物理模型)；

2． 掌握如何使用Java代码获得HBase连接，并熟练Java对HBase数据库的基本操作，进一步加深对HBase表概念的理解。

23.2 实验要求

通过Java代码实现与HBase数据库连接，然后用Java API创建HBase表，向创建的表中写数据，最后将表中数据读取出来并展示。

23.3 实验原理

逻辑模型：HBase以表的形式存储数据，每个表由行和列组成，每个列属于一个特定的列族(Column Family)。表中的行和列确定的存储单元称为一个元素(Cell)，每个元素保存了同一份数据的多个版本，由时间戳(Time Stamp)来标识。行健是数据行在表中的唯一标识，并作为检索记录的主键。在HBase中访问表中的行只有三种方式：通过单个行健访问、给定行键的范围扫描、全表扫描。行健可以是任意字符串，默认按字段顺序存储。表中的列定义为: <family>:<qualifier>(<列族>:<限定符>)，通过列族和限定符两部分可以唯一指定一个数据的存储列。元素由行健、列(<列族>:<限定符>)和时间戳唯一确定，元素中的数据以字节码的形式存储，没有类型之分。

物理模型：HBase是按照列存储的稀疏行/列矩阵，其物理模型实际上就是把概念模型中的一个行进行分割，并按照列族存储。

23.4 实验步骤

本实验主要演示HBase Java API的一些基本操作，包括取得链接，创建表，写数据，查询等几个步骤，具体内容如下：

首先，启动HBase集群(参考实验二十二)。

其次，从HBase安装包的lib目录导入如下jar包到开发工具(jar包的版本号以实际的安装中的版本号为主)：

commons-codec-1.4.jar  
 commons-collections-3.2.2.jar  
 commons-configuration-1.6.jar  
 commons-lang-2.6.jar  
 commons-logging-1.2.jar  
 guava-12.0.1.jar  
 hadoop-auth-2.7.2.jar  
 hadoop-common-2.7.2.jar  
 hadoop-hdfs-2.7.2.jar  
 hbase-client-1.1.2.jar  
 hbase-common-1.1.2.jar  
 hbase-protocol-1.1.2.jar  
 htrace-core-3.1.0-incubating.jar  
 httpclient-4.4.jar  
 httpcore-4.4.jar  
 libfb303-0.9.2.jar  
 log4j-1.2.17.jar  
 metrics-core-2.2.0.jar  
 netty-all-4.0.23.Final.jar  
 protobuf-java-2.5.0.jar  
 slf4j-api-1.7.7.jar  
 slf4j-log4j12-1.6.4.jar  
 zookeeper-3.4.6.jar

然后，获得HBase连接，代码实现：

Configuration configuration = HBaseConfiguration.create();

Connection connection;

configuration.set("hbase.zookeeper.quorum", "slave1:2181,slave2:2181,slave3:2181");

configuration.set("zookeeper.znode.parent", "/hbase");

connection = ConnectionFactory.createConnection(configuration);

然后，通过连接实现对HBase数据库的一些基本操作，如下：

新建HBase表，代码实现：

//获得HBaseAdmin对象

Admin admin = connection.getAdmin();

//表名称

String tn = "mytable";

TableName tableName = TableName.valueOf(tn);

//表不存在时创建表

if(!admin.tableExists(tableName))

{

//创建表描述对象

HTableDescriptor tableDescriptor = new HTableDescriptor(tableName);

//列簇1

HColumnDescriptor columnDescriptor1 = new HColumnDescriptor("c1".getBytes());

tableDescriptor.addFamily(columnDescriptor1);

//列簇2

HColumnDescriptor columnDescriptor2 = new HColumnDescriptor("c2".getBytes());

tableDescriptor.addFamily(columnDescriptor2);

//用HBaseAdmin对象创建表

admin.createTable(tableDescriptor);

}

//关闭HBaseAdmin对象

admin.close();

向表put数据，代码实现：

//获得table接口

Table table = connection.getTable(TableName.valueOf("mytable"));

//添加的数据对象集合

List<Put> putList = new ArrayList<Put>();

//添加10行数据

for(int i=0; i<10; i++)

{

//put对象(rowkey)

String rowkey = "mykey" + i;

Put put = new Put(rowkey.getBytes());

//列簇 , 列名, 值

put.addColumn("c1".getBytes(), "c1tofamily1".getBytes(), ("aaa"+i).getBytes());

put.addColumn("c1".getBytes(), "c2tofamily1".getBytes(), ("bbb"+i).getBytes());

put.addColumn("c2".getBytes(), "c1tofamily2".getBytes(), ("ccc"+i).getBytes());

putList.add(put);

}

table.put(putList);

table.close();

查询数据，代码实现：

//获得table接口（这行代码注意取舍，如果查询的代码和插入代码在同一个类中，则可以不要下面的这行）

Table table = connection.getTable(TableName.valueOf("mytable"));

//Scan 对象

Scan scan = new Scan();

//限定rowkey查询范围

scan.setStartRow("mykey0".getBytes());

scan.setStopRow("mykey9".getBytes());

//只查询c1：c1tofamily1列

scan.addColumn("c1".getBytes(), "c1tofamily1".getBytes());

//过滤器集合

FilterList filterList = new FilterList();

//查询符合条件c1：c1tofamily1==aaa7的记录

Filter filter1 = new SingleColumnValueFilter("c1".getBytes(), "c1tofamily1".getBytes(),

CompareFilter.CompareOp.EQUAL, "aaa7".getBytes());

filterList.addFilter(filter1);

scan.setFilter(filterList);

ResultScanner results = table.getScanner(scan);

for (Result result : results) {

System.out.println("获得到rowkey:" + new String(result.getRow()));

for (Cell cell : result.rawCells()) {

System.out.println("列簇：" +

Bytes.toString(cell.getFamilyArray(),cell.getFamilyOffset(),cell.getFamilyLength())

+ "列:" +

Bytes.toString(cell.getQualifierArray(),cell.getQualifierOffset(),cell.getQualifierLength())

+ "值:" +

Bytes.toString(cell.getValueArray(), cell.getValueOffset(), cell.getValueLength()));

}

}

results.close();

table.close();

最后，将Java代码打成jar包，并上传到客户端执行。

java -jar HbaseTest.jar

附件：完整代码如下：

**import** org.apache.hadoop.conf.Configuration;  
**import** org.apache.hadoop.hbase.\*;  
**import** org.apache.hadoop.hbase.client.\*;  
**import** org.apache.hadoop.hbase.filter.CompareFilter;  
**import** org.apache.hadoop.hbase.filter.Filter;  
**import** org.apache.hadoop.hbase.filter.FilterList;  
**import** org.apache.hadoop.hbase.filter.SingleColumnValueFilter;  
**import** org.apache.hadoop.hbase.util.Bytes;  
**import** java.io.IOException;  
**import** java.util.ArrayList;  
**import** java.util.List;  
**public class** Main {  
 **public static void** main(String[] args) {  
  
 Configuration configuration = HBaseConfiguration.*create*();  
 Connection connection;  
 configuration.set(**"hbase.zookeeper.quorum"**, **"slave1:2181,slave2:2181,slave3:2181"**);  
 configuration.set(**"zookeeper.znode.parent"**, **"/hbase"**);  
 **try** {  
 connection = ConnectionFactory.*createConnection*(configuration);  
 *//获得HBaseAdmin对象* Admin admin = connection.getAdmin();  
 *//表名称* String tn = **"mytable"**;  
 TableName tableName = TableName.*valueOf*(tn);  
 *//表不存在时创建表* **if** (!admin.tableExists(tableName)) {  
 *//创建表描述对象* HTableDescriptor tableDescriptor = **new** HTableDescriptor(tableName);  
 *//列簇1* HColumnDescriptor columnDescriptor1 = **new** HColumnDescriptor(**"c1"**.getBytes());  
 tableDescriptor.addFamily(columnDescriptor1);  
 *//列簇2* HColumnDescriptor columnDescriptor2 = **new** HColumnDescriptor(**"c2"**.getBytes());  
 tableDescriptor.addFamily(columnDescriptor2);  
 *//用HBaseAdmin对象创建表* admin.createTable(tableDescriptor);  
 }  
 *//关闭HBaseAdmin对象* admin.close();  
 *//向表put数据，代码实现：  
 //获得table接口* Table table = connection.getTable(TableName.*valueOf*(**"mytable"**));  
 *//添加的数据对象集合* List<Put> putList = **new** ArrayList<Put>();  
 *//添加10行数据* **for** (**int** i = 0; i < 10; i++) {  
 *//put对象(rowkey)* String rowkey = **"mykey"** + i;  
 Put put = **new** Put(rowkey.getBytes());  
 *//列簇 , 列名, 值* put.addColumn(**"c1"**.getBytes(), **"c1tofamily1"**.getBytes(), (**"aaa"** + i).getBytes());  
 put.addColumn(**"c1"**.getBytes(), **"c2tofamily1"**.getBytes(), (**"bbb"** + i).getBytes());  
 put.addColumn(**"c2"**.getBytes(), **"c1tofamily2"**.getBytes(), (**"ccc"** + i).getBytes());  
 putList.add(put);  
 }  
 table.put(putList);  
 table.close();  
 *//查询数据，代码实现：  
 //获得table接口  
// Table table = connection.getTable(TableName.valueOf("mytable"));  
 //Scan 对象* Scan scan = **new** Scan();  
 *//限定rowkey查询范围* scan.setStartRow(**"mykey0"**.getBytes());  
 scan.setStopRow(**"mykey9"**.getBytes());  
 *//只查询c1：c1tofamily1列* scan.addColumn(**"c1"**.getBytes(), **"c1tofamily1"**.getBytes());  
 *//过滤器集合* FilterList filterList = **new** FilterList();  
 *//查询符合条件c1：c1tofamily1==aaa7的记录* Filter filter1 = **new** SingleColumnValueFilter(**"c1"**.getBytes(), **"c1tofamily1"**.getBytes(), CompareFilter.CompareOp.***EQUAL***, **"aaa7"**.getBytes());  
 filterList.addFilter(filter1);  
 scan.setFilter(filterList);  
 ResultScanner results = table.getScanner(scan);  
 **for** (Result result : results) {  
 System.***out***.println(**"获得到rowkey:"** + **new** String(result.getRow()));  
 **for** (Cell cell : result.rawCells()) {  
 System.***out***.println(**"列簇："** +  
 Bytes.*toString*(cell.getFamilyArray(), cell.getFamilyOffset(), cell.getFamilyLength())  
 + **"列:"** +  
 Bytes.*toString*(cell.getQualifierArray(), cell.getQualifierOffset(), cell.getQualifierLength())  
 + **"值:"** +  
 Bytes.*toString*(cell.getValueArray(), cell.getValueOffset(), cell.getValueLength()));  
 }  
 }  
 results.close();  
 table.close();  
  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 }  
}

23.5 实验结果

表创建完，然后添加数据后，可以通过shell查看mytable表数据，共插入10条数据，数据内容如图23-1所示。

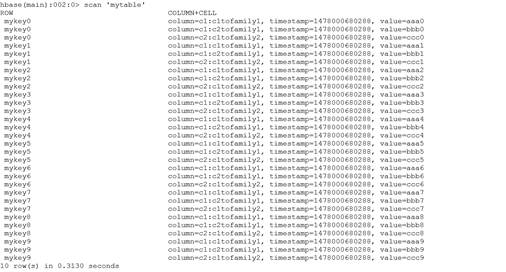


图23-1 HBase Shell界面图

客户端提交jar包后，运行Java代码打印的日志信息如图23-2所示(日志中可能会出现乱码，这个没有关系，不影响实验结果的查看)。

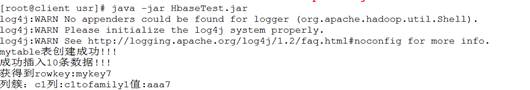


图23-2 Java代码执行结果图

实验二十四 部署Storm

24.1 实验目的

1． 掌握Storm基础简介及体系架构；

2． 掌握Storm集群安装部署；

3． 掌握Storm和Zookeeper之间的关系，并加深对Storm架构和原理的理解。

24.2 实验要求

1． 巩固学习下实验一、实验二十；

2． 部署四个节点的Storm集群，以master节点作为主节点，其他三个slave节点作为从节点，并修改Storm Web的端口为8081，并引用外部Zookeeper。

24.3 实验原理

Storm简介：Storm是一个分布式的、高容错的基于数据流的实时处理系统，可以简单、可靠的处理大量的数据流。Storm支持水平扩展，具有高容错性，保证每个消息都会得到处理，而且处理速度很快（在一个小集群中，每个结点每秒可以处理数以百万计的消息），它有以下特点：编程模型简单、可扩展、高可靠性、高容错性、支持多种编程语言、支持本地模式、高效。Storm有很多使用场景：如实时分析，在线机器学习，持续计算，分布式RPC，ETL等等。

体系架构：Storm共有两层体系结构，第一层采用master/slave架构，第二层为DAG流式处理器，第一层资源管理器主要负责管理集群资源、响应和调度用户任务，第二层流式处理器则实际执行用户任务

集群资源管理层：Storm的集群资源管理器采用master/slave架构，主节点即控制节点(master node)和从节点即工作节点(worker node)。控制节点上面运行一个叫Nimbus后台服务程序,它的作用类似Hadoop里面的JobTracker， Nimbus负责在集群里面分发代码，分配计算任务给机器，并且监控状态。每一个工作节点上面运行一个叫做Supervisor的服务程序。Supervisor会监听分配给它那台机器的工作，根据需要启动/关闭工作进程worker。每一个工作进程执行一个topology的一个子集；一个运行的topology由运行在很多机器上的很多工作进程worker组成。(一个supervisor里面有多个worker，一个worker是一个JVM。可以配置worker的数量，对应的是conf/storm.yaml中的supervisor.slot的数量），架构图如图24-1所示。

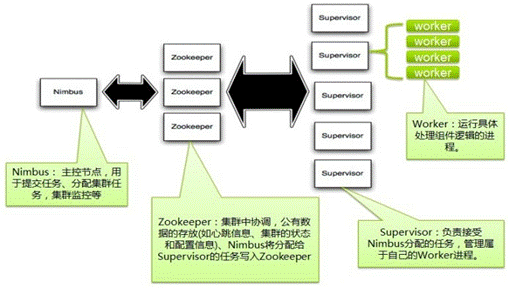


图24-1 架构图

称集群信息(Nimbus协议、Supervisor节点位置) 、任务分配信息等关键数据为元数据。Storm使用ZooKeeper集群来共享元数据，这些元数据对Storm非常重要，比如Nimbus通过这些元数据感知Supervisor节点，Supervisor通过Zookeeper集群感知任务分配情况。Nimbus和Supervisor之间的所有协调工作都是通过Zookeeper集群完成。另外，Nimbus进程和Supervisor进程都是快速失败(fail-fast)和无状态的｡所有的状态要么在zookeeper里面, 要么在本地磁盘上。这也就意味着你可以用kill -9来杀死Nimbus和Supervisor进程,然后再重启它们，就好像什么都没有发生过，这个设计使得Storm异常的稳定。

数据模型：Storm实现了一种数据流模型，其中数据持续地流经一个转换实体网络。一个数据流的抽象称为一个流（stream），这是一个无限的元组序列。元组（tuple）就像一种使用一些附加的序列化代码来表示标准数据类型（比如整数、浮点和字节数组）或用户定义类型的结构。每个流由一个唯一ID定义，这个ID可用于构建数据源和接收器（sink）的拓扑结构。流起源于喷嘴（spout），Spout将数据从外部来源流入 Storm 拓扑结构中。接收器（或提供转换的实体）称为螺栓（bolt）。螺栓实现了一个流上的单一转换和一个 Storm 拓扑结构中的所有处理。Bolt既可实现 MapReduce之类的传统功能，也可实现更复杂的操作（单步功能），比如过滤、聚合或与数据库等外部实体通信。典型的 Storm 拓扑结构会实现多个转换，因此需要多个具有独立元组流的Bolt。Bolt和Spout都实现为Linux系统中的一个或多个任务。

24.4 实验步骤

本实验主要演示Storm集群的安装部署，Storm依赖于Zookeeper，所以该实验大致可分为部署Zookeeper、部署Storm、启动Storm集群三个大步骤。

首先，配置SSH无密钥登录(参考实验1)。

然后，安装Zookeeper集群(参考实验20)。

最后，部署Storm，具体内容如下：

将Storm 安装包解压到/usr/cstor目录，并将Storm解压目录所属用户改成root:root。

[root@master ~]tar -zxvf apache-storm-0.10.0.tar.gz -c /usr/cstor

[root@master ~]mv /usr/cstor/apache-storm-0.10.0 /usr/cstor/storm

[root@master ~]chown -R root:root /usr/cstor/storm

进入解压目录下，把conf目录下的storm.yaml修改和添加配置项目（每个配置项前面必须留有空格，否则会无法识别），如下：

Storm集群使用的Zookeeper集群地址，其格式如下：

storm.zookeeper.servers:

- "slave1"

- "slave2"

- "slave3"

本地存储目录(如果目录不存在记得手动创建: mkdir -p /usr/cstor/storm/workdir )：

storm.local.dir: "/usr/cstor/storm/workdir"

Storm集群Nimbus机器地址：

nimbus.host: "master"

对于每个Supervisor工作节点，需要配置该工作节点可以运行的worker数量。每个worker占用一个单独的端口用于接收消息，该配置选项即用于定义哪些端口是可被worker使用的。默认情况下，每个节点上可运行4个workers，分别在6700、6701、6702和6703端口，如：

supervisor.slots.ports:

- 6700

- 6701

- 6702

- 6703

UI端口（web端口，默认8080）

ui.port: 8081

将/usr/cstor/storm目录传到另外三个节点上。

[root@master ~]scp -r /usr/cstor/storm hadoop@slave1:/usr/cstor

[root@master ~]scp -r /usr/cstor/storm hadoop@slave2:/usr/cstor

[root@master ~]scp -r /usr/cstor/storm hadoop@slave3:/usr/cstor

然后，启动Storm集群(保证Zookeeper在此之前已启动)，启动步骤如下：

主节点（master）启动nimbus服务。

cd /usr/cstor/storm/bin

nohup ./storm nimbus >/dev/null 2>&1 &

从节点（3个）启动supervisor服务。

cd /usr/cstor/storm/bin

nohup ./storm supervisor >/dev/null 2>&1 &

主节点（master）启动ui服务。

cd /usr/cstor/storm/bin

nohup ./storm ui >/dev/null 2>&1 &

24.5 实验结果

在Storm集群主节点执行jps命令查看Java进程，有nimbus进程和core进程，其中nimbus进程为Storm主节点进程，core为WEB进程，如图24-2所示。

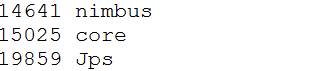


图24-2 Storm主节点进程图

在Storm集群从节点执行jps命令查看Java进程，有supervisor进程，此进程为Storm从节点的进程，如图24-3所示。

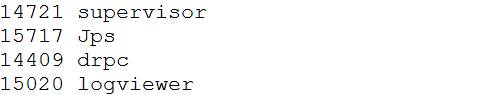


图24-3 Storm从节点进程图

启动好Storm集群后。可以通过流浪器访问Storm WEB页面（http://master:8081），查看Storm集群的一些基本情况，如图24-4所示。

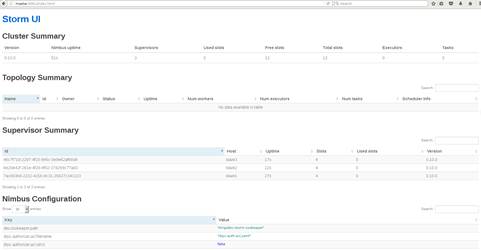


图24-4 Storm WEB页面图

实验二十五 实时WordCountTopology

25.1 实验目的

掌握如何用Java代码来实现Storm任务的拓扑，掌握一个拓扑中Spout和Bolt的关系及如何组织它们之间的关系，掌握如何将Storm任务提交到集群。

25.2 实验要求

编写一个Storm拓扑，一个Spout每个一秒钟随机生成一个单词并发射给Bolt，Bolt统计接收到的每个单词出现的频率并每隔一秒钟实时打印一次统计结果，最后将任务提交到集群运行，并通过日志查看任务运行结果。

25.3 实验原理

Storm集群和Hadoop集群表面上看很类似。但是Hadoop上运行的是MapReduce jobs，而在Storm上运行的是拓扑（topology），这两者之间是非常不一样的。一个关键的区别是： 一个MapReduce job最终会结束， 而一个topology永远会运行（除非你手动kill掉）。

25.3.1 Topologies

一个topology是spouts和bolts组成的图，通过stream groupings将图中的spouts和bolts连接起来，如图25-1所示。

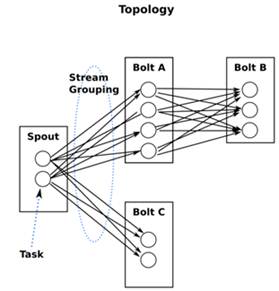


图25-1 Topologies运行图

一个topology会一直运行直到你手动kill掉，Storm自动重新分配执行失败的任务， 并且Storm可以保证你不会有数据丢失（如果开启了高可靠性的话）。如果一些机器意外停机它上面的所有任务会被转移到其他机器上。

运行一个topology很简单。首先，把你所有的代码以及所依赖的jar打进一个jar包。然后运行类似下面的这个命令：

storm jar all-my-code.jar backtype.storm.MyTopology arg1 arg2

这个命令会运行主类: backtype.strom.MyTopology, 参数是arg1, arg2。这个类的main函数定义这个topology并且把它提交给Nimbus。storm jar负责连接到Nimbus并且上传jar包。

Topology的定义是一个Thrift结构，并且Nimbus就是一个Thrift服务， 你可以提交由任何语言创建的topology。上面的方面是用JVM-based语言提交的最简单的方法。

25.3.2 Spouts

消息源spout是Storm里面一个topology里面的消息生产者。一般来说消息源会从一个外部源读取数据并且向topology里面发出消息：tuple。Spout可以是可靠的也可以是不可靠的。如果这个tuple没有被storm成功处理，可靠的消息源spouts可以重新发射一个tuple， 但是不可靠的消息源spouts一旦发出一个tuple就不能重发了。

消息源可以发射多条消息流stream。使用OutputFieldsDeclarer.declareStream来定义多个stream，然后使用SpoutOutputCollector来发射指定的stream。

Spout类里面最重要的方法是nextTuple。要么发射一个新的tuple到topology里面或者简单的返回如果已经没有新的tuple。要注意的是nextTuple方法不能阻塞，因为storm在同一个线程上面调用所有消息源spout的方法。

另外两个比较重要的spout方法是ack和fail。storm在检测到一个tuple被整个topology成功处理的时候调用ack，否则调用fail。storm只对可靠的spout调用ack和fail。

25.3.3 Bolts

所有的消息处理逻辑被封装在bolts里面。Bolts可以做很多事情：过滤，聚合，查询数据库等等。

Bolts可以简单的做消息流的传递。复杂的消息流处理往往需要很多步骤，从而也就需要经过很多bolts。比如算出一堆图片里面被转发最多的图片就至少需要两步：第一步算出每个图片的转发数量。第二步找出转发最多的前10个图片。（如果要把这个过程做得更具有扩展性那么可能需要更多的步骤）。

Bolts可以发射多条消息流， 使用OutputFieldsDeclarer.declareStream定义stream，使用OutputCollector.emit来选择要发射的stream。

Bolts的主要方法是execute, 它以一个tuple作为输入，bolts使用OutputCollector来发射tuple，bolts必须要为它处理的每一个tuple调用OutputCollector的ack方法，以通知Storm这个tuple被处理完成了，从而通知这个tuple的发射者spouts。 一般的流程是： bolts处理一个输入tuple, 发射0个或者多个tuple, 然后调用ack通知storm自己已经处理过这个tuple了。storm提供了一个IBasicBolt会自动调用ack。

25.4 实验步骤

本实验主要演示一个完整的Storm拓扑编码过程，主要包含Spout、Bolt和构建Topology几个步骤。

首先，启动Storm集群。

其次，将Storm安装包的lib目录内如下jar包导入到开发工具： 

图25-2 相关jar包列表

然后，编写代码，实现一个完整的Topology，内容如下：

Spout随机发送单词，代码实现：

package cproc.word;

import java.util.Map;

import java.util.Random;

import backtype.storm.spout.SpoutOutputCollector;

import backtype.storm.task.TopologyContext;

import backtype.storm.topology.OutputFieldsDeclarer;

import backtype.storm.topology.base.BaseRichSpout;

import backtype.storm.tuple.Fields;

import backtype.storm.tuple.Values;

import backtype.storm.utils.Utils;

public class WordReaderSpout extends BaseRichSpout {

private SpoutOutputCollector collector;

@Override

public void open(Map conf, TopologyContext context, SpoutOutputCollector collector)

{

this.collector = collector;

}

@Override

public void nextTuple() {

//这个方法会不断被调用，为了降低它对CPU的消耗，让它sleep一下

Utils.sleep(1000);

final String[] words = new String[] {"nathan", "mike", "jackson", "golda", "bertels"};

Random rand = new Random();

String word = words[rand.nextInt(words.length)];

collector.emit(new Values(word));

}

@Override

public void declareOutputFields(OutputFieldsDeclarer declarer) {

declarer.declare(new Fields("word"));

}

}

Bolt单词计数，并每隔一秒打印一次，代码实现：

package cproc.word;

import java.util.HashMap;

import java.util.Map;

import java.util.Map.Entry;

import backtype.storm.task.TopologyContext;

import backtype.storm.topology.BasicOutputCollector;

import backtype.storm.topology.OutputFieldsDeclarer;

import backtype.storm.topology.base.BaseBasicBolt;

import backtype.storm.tuple.Tuple;

public class WordCounterBolt extends BaseBasicBolt {

private static final long serialVersionUID = 5683648523524179434L;

private HashMap<String, Integer> counters = new HashMap<String, Integer>();

private volatile boolean edit = false;

@Override

public void prepare(Map stormConf, TopologyContext context) {

//定义一个线程1秒钟打印一次统计的信息

new Thread(new Runnable() {

public void run() {

while (true) {

if (edit) {

for (Entry<String, Integer> entry : counters.entrySet())

{

System.out.println(entry.getKey() + " : " + entry.getValue());

}

edit = false;

}

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}).start();

}

@Override

public void execute(Tuple input, BasicOutputCollector collector) {

String str = input.getString(0);

if (!counters.containsKey(str)) {

counters.put(str, 1);

} else {

Integer c = counters.get(str) + 1;

counters.put(str, c);

}

edit = true;

}

@Override

public void declareOutputFields(OutputFieldsDeclarer declarer) {

}

}

构建Topology并提交到集群主函数，代码实现：

package cproc.word;

import backtype.storm.Config;

import backtype.storm.StormSubmitter;

import backtype.storm.generated.AlreadyAliveException;

import backtype.storm.generated.AuthorizationException;

import backtype.storm.generated.InvalidTopologyException;

import backtype.storm.topology.TopologyBuilder;

public class WordCountTopo {

public static void main(String[] args) throws Exception{

//构建Topology

TopologyBuilder builder = new TopologyBuilder();

builder.setSpout("word-reader", new WordReaderSpout());

builder.setBolt("word-counter", new WordCounterBolt())

.shuffleGrouping("word-reader");

Config conf = new Config();

//集群方式提交

StormSubmitter.submitTopologyWithProgressBar("wordCount", conf,

builder.createTopology());

}

}

然后，将Storm代码打成wordCount-Storm.jar(打包的时候不要包含storm中的jar，不然会报错的，将无法运行，即：wordCount-Storm.jar中只包含上面三个类的代码)上传到主节点的/usr/cstor/storm/bin目录下，在主节点进入Storm安装目录的bin下面用以下命令提交任务：

cd /usr/cstor/storm/bin

./storm jar wordCount-Storm.jar cproc.word.WordCountTopo wordCount

使用以下命令结束storm任务：

./storm kill wordCount

25.5 实验结果

Storm任务执行时，可以查看Storm 日志文件，日志里面打印了统计的单词结果，日志内容如图25-2所示。

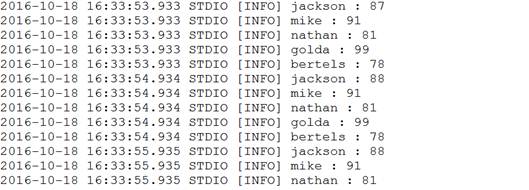


图25-3 Storm任务日志图

实验二十六 文件数据Flume至HDFS

26.1 实验目的

1． 掌握Flume的安装部署；

2． 掌握一个agent中source、sink、channel组件之间的关系；

3． 加深对Flume结构和概念的理解；

4． 掌握Flume的编码方法及启动任务方法。

26.2 实验要求

1． 在一台机器上(本例以slave1为例)部署Flume；

2． 实时收集本地hadoop的日志的最新信息然后将收集到日志信息以一分钟一个文件的形式写入HDFS目录中。

26.3 实验原理

Flume是Cloudera提供的一个高可用的，高可靠的，分布式的海量日志采集、聚合和传输的系统，Flume支持在日志系统中定制各类数据发送方，用于收集数据；同时，Flume提供对数据进行简单处理，并写到各种数据接受方（可定制）的能力。

Flume提供对数据进行简单处理，并写到各种数据接受方（可定制）的能力 Flume提供了从console（控制台）、RPC（Thrift-RPC）、text（文件）、tail（UNIX tail）、syslog（syslog日志系统，支持TCP和UDP等2种模式），exec（命令执行）等数据源上收集数据的能力。

当前Flume有两个版本Flume 0.9X版本的统称Flume-og，Flume1.X版本的统称Flume-ng。由于Flume-ng经过重大重构，与Flume-og有很大不同，使用时请注意区分。

Flume-og采用了多Master的方式。为了保证配置数据的一致性，Flume引入了ZooKeeper，用于保存配置数据，ZooKeeper本身可保证配置数据的一致性和高可用，另外，在配置数据发生变化时，ZooKeeper可以通知Flume Master节点。Flume Master间使用gossip协议同步数据。

Flume-ng最明显的改动就是取消了集中管理配置的 Master 和 Zookeeper，变为一个纯粹的传输工具。Flume-ng另一个主要的不同点是读入数据和写出数据现在由不同的工作线程处理（称为Runner）。 在 Flume-og 中，读入线程同样做写出工作（除了故障重试）。如果写出慢的话（不是完全失败），它将阻塞 Flume 接收数据的能力。这种异步的设计使读入线程可以顺畅的工作而无需关注下游的任何问题。

Flume以agent为最小的独立运行单位。一个agent就是一个JVM。单agent由Source、Sink和Channel三大组件构成，如图26-1所示。

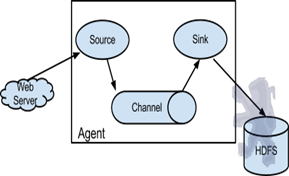


图26-1 Flume运行图

值得注意的是，Flume提供了大量内置的Source、Channel和Sink类型。不同类型的Source,Channel和Sink可以自由组合。组合方式基于用户设置的配置文件，非常灵活。比如：Channel可以把事件暂存在内存里，也可以持久化到本地硬盘上。Sink可以把日志写入HDFS, HBase，甚至是另外一个Source等等。Flume支持用户建立多级流，也就是说，多个agent可以协同工作，并且支持Fan-in、Fan-out、Contextual Routing、Backup Routes，这也正是强大之处。如图26-2所示：

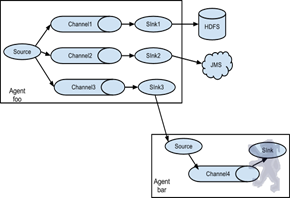


图26-2 Source、Channel和Sink类型示意图

26.3.1 flume的特点

flume是一个分布式、可靠、和高可用的海量日志采集、聚合和传输的系统。支持在日志系统中定制各类数据发送方，用于收集数据;同时，Flume提供对数据进行简单处理，并写到各种数据接受方(比如文本、HDFS、Hbase等)的能力 。

flume的数据流由事件(Event)贯穿始终。事件是Flume的基本数据单位，它携带日志数据(字节数组形式)并且携带有头信息，这些Event由Agent外部的Source生成，当Source捕获事件后会进行特定的格式化，然后Source会把事件推入(单个或多个)Channel中。你可以把Channel看作是一个缓冲区，它将保存事件直到Sink处理完该事件。Sink负责持久化日志或者把事件推向另一个Source。

26.3.2 flume的可靠性

当节点出现故障时，日志能够被传送到其他节点上而不会丢失。Flume提供了三种级别的可靠性保障，从强到弱依次分别为：end-to-end（收到数据agent首先将event写到磁盘上，当数据传送成功后，再删除；如果数据发送失败，可以重新发送。），Store on failure（这也是scribe采用的策略，当数据接收方crash时，将数据写到本地，待恢复后，继续发送），Besteffort（数据发送到接收方后，不会进行确认）。

26.4 实验步骤

本实验主要演示Flume安装，及启动一个Flume收集日志信息的例子，实验主要包含以下三个大步骤。

首先，启动Hadoop集群。

其次，(剩下的所有步骤只需要在master上操作就可以了)安装并配置Flume任务，内容如下：

将Flume 安装包解压到/usr/cstor目录，并将flume目录所属用户改成root:root。

tar -zxvf flume-1.5.2.tar.gz -c /usr/cstor

chown -R root:root /usr/cstor/flume

进入解压目录下，在conf目录下新建test.conf文件并添加以下配置内容：

#定义agent中各组件名称

agent1.sources=source1

agent1.sinks=sink1

agent1.channels=channel1

# source1组件的配置参数

agent1.sources.source1.type=exec

#此处的文件/home/source.log需要手动生成，见后续说明

agent1.sources.source1.command=tail -n +0 -F /home/source.log

# channel1的配置参数

agent1.channels.channel1.type=memory

agent1.channels.channel1.capacity=1000

agent1.channels.channel1.transactionCapactiy=100

# sink1的配置参数

agent1.sinks.sink1.type=hdfs

agent1.sinks.sink1.hdfs.path=hdfs://master:8020/flume/data

agent1.sinks.sink1.hdfs.fileType=DataStream

#时间类型

agent1.sinks.sink1.hdfs.useLocalTimeStamp=true

agent1.sinks.sink1.hdfs.writeFormat=TEXT

#文件前缀

agent1.sinks.sink1.hdfs.filePrefix=%Y-%m-%d-%H-%M

#60秒滚动生成一个文件

agent1.sinks.sink1.hdfs.rollInterval=60

#HDFS块副本数

agent1.sinks.sink1.hdfs.minBlockReplicas=1

#不根据文件大小滚动文件

agent1.sinks.sink1.hdfs.rollSize=0

#不根据消息条数滚动文件

agent1.sinks.sink1.hdfs.rollCount=0

#不根据多长时间未收到消息滚动文件

agent1.sinks.sink1.hdfs.idleTimeout=0

# 将source和sink 绑定到channel

agent1.sources.source1.channels=channel1

agent1.sinks.sink1.channel=channel1

然后，在HDFS上创建/flume/data目录：

cd /usr/cstor/hadoop/bin

./hdfs dfs -mkdir /flume

./hdfs dfs -mkdir /flume/data

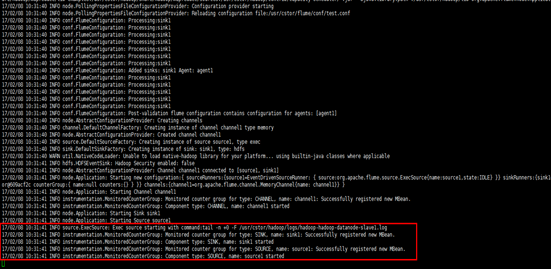
最后，进入Flume安装的bin目录下

cd /usr/cstor/flume/bin

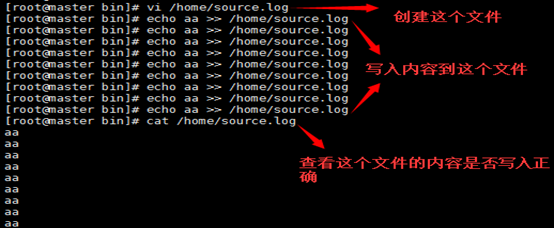
启动Flume，开始收集日志信息。

https://bd.cstor.cn/experiment/exp26.files/image003.png

看到如下结果就表示启动成功：



启动成功之后需要手动生成消息源即配置文件中的/home/source.log，使用如下命令去不断的写入文字到/home/source.log中：



到此就可以去查看实验结果了

26.5 实验结果

实验结果查看的步骤：

https://bd.cstor.cn/experiment/exp26.files/image006.png

图26-3 Flume写入HDFS的文件列表图

实验二十七 Kafka订阅推送示例

27.1 实验目的

1． 掌握Kafka的安装部署；

2． 掌握Kafka的topic创建及如何生成消息和消费消息；

3． 掌握Kafka和Zookeeper之间的关系；

4． 了解Kafka如何保存数据及加深对Kafka相关概念的理解。

27.2 实验要求

在三台机器上(以slave1，slave2，slave3为例)，分别部署一个broker，Zookeeper使用的是单独的集群，然后创建一个topic，启动模拟的生产者和消费者脚本，在生产者端向topic里写数据，在消费者端观察读取到的数据。

27.3 实验原理

27.3.1 Kafka简介

Kafka是一种高吞吐量的分布式发布订阅消息系统，它可以处理消费者规模的网站中的所有动作流数据。 它提供了类似于JMS的特性，但是在设计实现上完全不同，此外它并不是JMS规范的实现。kafka对消息保存时根据Topic进行归类，发送消息者成为Producer,消息接受者成为Consumer,此外kafka集群有多个kafka实例组成，每个实例(server)成为broker。无论是kafka集群，还是producer和consumer都依赖于zookeeper来保证系统可用性集群保存一些meta信息。如图27-1所示：

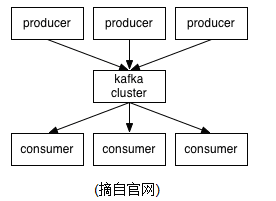


图27-1 Kafka工作示意图

一个Topic的多个partitions,被分布在kafka集群中的多个server上;每个server(kafka实例)负责partitions中消息的读写操作;此外kafka还可以配置partitions需要备份的个数(replicas),每个partition将会被备份到多台机器上,以提高可用性。

基于replicated方案,那么就意味着需要对多个备份进行调度;每个partition都有一个server为"leader";leader负责所有的读写操作,如果leader失效,那么将会有其他follower来接管(成为新的leader);follower只是单调的和leader跟进,同步消息即可..由此可见作为leader的server承载了全部的请求压力,因此从集群的整体考虑,有多少个partitions就意味着有多少个"leader",kafka会将"leader"均衡的分散在每个实例上,来确保整体的性能稳定。

生产者：Producer将消息发布到指定的Topic中,同时Producer也能决定将此消息归属于哪个partition;比如基于"round-robin"方式或者通过其他的一些算法等。

消费者：本质上kafka只支持Topic.每个consumer属于一个consumer group;反过来说,每个group中可以有多个consumer.发送到Topic的消息,只会被订阅此Topic的每个group中的一个consumer消费。

如果所有的consumer都具有相同的group,这种情况和queue模式很像;消息将会在consumers之间负载均衡。

如果所有的consumer都具有不同的group,那这就是"发布-订阅";消息将会广播给所有的消费者。

在kafka中,一个partition中的消息只会被group中的一个consumer消费;每个group中consumer消息消费互相独立;我们可以认为一个group是一个"订阅"者,一个Topic中的每个partions,只会被一个"订阅者"中的一个consumer消费,不过一个consumer可以消费多个partitions中的消息.kafka只能保证一个partition中的消息被某个consumer消费时,消息是顺序的.事实上,从Topic角度来说,消息仍不是有序的。

kafka的设计原理决定,对于一个topic,同一个group中不能有多于partitions个数的consumer同时消费,否则将意味着某些consumer将无法得到消息。

Guarantees

（1）发送到partitions中的消息将会按照它接收的顺序追加到日志中。

（2）对于消费者而言,它们消费消息的顺序和日志中消息顺序一致。

（3）如果Topic的"replicationfactor"为N,那么允许N-1个kafka实例失效。

27.3.2 Kafka使用场景

（1）Messaging

对于一些常规的消息系统,kafka是个不错的选择;partitons/replication和容错,可以使kafka具有良好的扩展性和性能优势.不过到目前为止,我们应该很清楚认识到,kafka并没有提供JMS中的"事务性""消息传输担保(消息确认机制)""消息分组"等企业级特性;kafka只能使用作为"常规"的消息系统,在一定程度上,尚未确保消息的发送与接收绝对可靠(比如,消息重发,消息发送丢失等)。

（2）Websit activity tracking

kafka可以作为"网站活性跟踪"的最佳工具;可以将网页/用户操作等信息发送到kafka中.并实时监控,或者离线统计分析等。

（3）Log Aggregation

kafka的特性决定它非常适合作为"日志收集中心";application可以将操作日志"批量""异步"的发送到kafka集群中,而不是保存在本地或者DB中;kafka可以批量提交消息/压缩消息等,这对producer端而言,几乎感觉不到性能的开支.此时consumer端可以使hadoop等其他系统化的存储和分析系统。

27.4 实验步骤

本实验主要演示Kafka的安装，及简单使用，Kafka数据保存在Zookeeper上，所以该实验主要包含以下三个步骤。

27.4.1 安装Zookeeper集群

首先，安装Zookeeper集群, 这个参照前面的实验操作，这里不再赘述。

27.4.2 安装Kafka集群

将Kafka 安装包解压到slave1的/usr/cstor目录，将kafka目录所属用户改成root:root，并将kafka目录传到其他两台机器上。

tar -zxvf kafka\_2.10-0.9.0.1.tar.gz -c /usr/cstor

chown -R root:root /usr/cstor/kafka

scp -r /usr/cstor/kafka hadoop@slave2:/usr/cstor

scp -r /usr/cstor/kafka hadoop@slave3:/usr/cstor

三台机器上分别进入解压目录下，在config目录修改server.properties文件，修改内容如下：

slave1配置：

#broker.id

broker.id=1

#broker.port

port=9092

#host.name

host.name=slave1

#本地日志文件位置

log.dirs=/usr/cstor/kafka/logs

#Zookeeper地址

zookeeper.connect=slave1:2181,slave2:2181,slave3:2181

slave2配置：

#broker.id

broker.id=2

#broker.port

port=9092

#host.name

host.name=slave2

#本地日志文件位置

log.dirs=/usr/cstor/kafka/logs

#Zookeeper地址

zookeeper.connect=slave1:2181,slave2:2181,slave3:2181

slave3配置：

#broker.id

broker.id=3

#broker.port

port=9092

#host.name

host.name=slave3

#本地日志文件位置

log.dirs=/usr/cstor/kafka/logs

#Zookeeper地址

zookeeper.connect=slave1:2181,slave2:2181,slave3:2181

然后，启动Kafka，并验证Kafka功能：

进入安装目录下的bin目录，三台机器上分别执行以下命令启动各自的Kafka服务：

cd /usr/cstor/kafka/bin

nohup ./kafka-server-start.sh ../config/server.properties &

在任意一台机器上，执行以下命令(以下三行命令不要换行，是一整行)创建topic：

./kafka-topics.sh --create

--zookeeper slave1:2181,slave2:2181,slave3:2181

--replication-factor 2 --partitions 2 --topic test

在任意一台机器上，执行以下命令(以下三行命令不要换行，是一整行)启动模拟producer：

./kafka-console-producer.sh

--broker-list slave1:9092,slave2:9092,slave3:9092

--topic test

在另一台机器上，执行以下命令(以下三行命令不要换行，是一整行)启动模拟consumer：

./kafka-console-consumer.sh

--zookeeper slave1:2181,slave2:2181,slave3:2181

--topic test --from-beginning

27.4.3 验证消息推送

在producer端输入任意信息，然后观察consumer端接收到的数据，如：

This is Kafka producer

Hello, Kafka

27.5 实验结果

启动producer端和Consumer端后，在Kafka producer端发送消息后，在Consumer端收到信息内容如图27-2所示。

https://bd.cstor.cn/experiment/exp27.files/image002.jpg

图27-2 Kafka接收信息图

实验二十八 Pig版WordCount

28.1 实验目的

掌握Pig的安装部署，了解Pig和MapReduce之间的关系，掌握Pig Latin 编程语言，加深对Pig相关概念的理解。

28.2 实验要求

1． 在集群中任一节点部署Pig（以下示例中选主节点）；

2． 在Pig的Hadoop模式下统计任意一个HDFS文本中的单词出现次数，并将统计结果打印出来；

3． 用Pig查询统计HDFS文件系统中结构化的数据。

28.3 实验原理

Pig 是一种探索大规模数据集的 脚本语言。MapReducer 的一个主要的 缺点 就是开发的 周期太长 了。我们要编写mapper和reducer，然后对代码进行编译打出jar 包，提交到本地的 JVM 或者是 hadoop的集群 上，最后获取结果，这个周期是非常 耗时 的，即使使用 Streaming (它是hadoop的一个工具，用来创建和运行一类特殊的map/reduce作业。所谓的特殊的map/reduce作业可以是可执行文件或脚本本件（python、PHP、c等）。Streaming使用“标准输入”和“标准输出”与我们编写的Map和Reduce进行数据的交换。由此可知，任何能够使用“标准输入”和“标准输出”的编程语言都可以用来编写MapReduce程序，能在这个过程中除去代码的编译和打包的步骤，但是这一个过程还是很耗时， Pig的强大之处 就是他只要 几行Pig Latin代码就能处理TB级别的数据 。Pig提供了多个命令用于 检查和处理程序中的数据结构 ，因此它能很好的支持我们写 查询 。Pig的一个很有用的特性就是它支持在输入数据中有代表性的一个 小的数据集上试运行。 所以。我们在处理大的数据集前可以用那一个小的数据集检查 我们的程序是不是有错误的。

Pig为大型的数据集的处理提供了更高层次的抽象。 MapReducer 能够让我们自己定义 连续执行的map和reduce函数 ，但是数据处理往往需要很多的MapReducer过程才能实现，所以将数据处理要求改写成MapReducer模式是很 复杂的 。和MapReducer相比，Pig提供了更加 丰富的数据结构 ，一般都是 多值 和 嵌套 的数据结构。Pig还提供了一套更强大的 数据交换操作 ，包括了MapReducer中被忽视的" join"操作。

Pig 被设计为可以 扩展 的，处理路径上的每一个部分， 载入 ， 存储 ， 过滤， 分组 ， 连接 ，都是可以 定制 的，这些操作都可以使用用户定义函数（user-defined function, UDF ）进行 修改 ，这些函数作用于Pig的嵌套数据模型。因此，它们可以在底层与Pig的操作集成， UDF的另外的一个好处是它们比MapReducer程序开发的库更易于重用。

但是，Pig并不适合处理所有的“数据处理”任务。和MapReducer一样，它是为数据 批处理 而设计的，如果想执行的查询只涉及一个大型数据集的一小部分数据，Pig的实现不是很好， 因为它要扫描整个数据集或其中的很大一部分。

Pig Latin程序是由一系列的" 操作"(operation)或"变换"(transformation)组成。每个操作或变换对输入进行 数据处理 ，然后产生 输出的结果 。这些操作整体上描述了一个 数据流 ，Pig执行的环境把数据流翻译为可执行的内部表示，并运行它。在Pig的内部，这些变换和操作被转换成一系列的MapReducer，但是我们一般情况下并不知道这些转换是怎么进行的， 我们的主要的精力就花在数据上，而不是执行的细节上面。

在有些情况下，Pig的表现不如MapReducer程序。总结为一句就是要么话花大量的时间来优化Java MapReducer程序，要么使用Pig Latin来编写查询确实能 节约时间。

28.4 实验步骤

本实验只要演示Pig的安装部署和Pig在Hadoop模式下的WordCount和结构化数据查询统计示例演示，只要包含以下三个步骤。

将Pig 安装包解压到/usr/cstor目录。

[root@master usr]# tar -zxvf pig-0.15.0.tar.gz

[root@master usr]# mv pig-0.15.0 /usr/cstor

tar -zxvf pig-0.15.0.tar.gz -c /usr/cstor

进入解压目录下，修改bin目录下pig文件，添加如下内容：

#指定Java安装路径

JAVA\_HOME=/usr/local/jdk1.7.0\_79

#指定Hadoop安装路径

HADOOP\_HOME=/usr/cstor/hadoop

#指定Hadoop配置文件的路径

PIG\_CLASSPATH=/usr/cstor/hadoop/etc/hadoop

Pig配置完成后，在安装Pig的节点进入hadoop安装目录下的sbin目录，并启动historyserver服务：

[root@master hadoop]# cd /usr/cstor/hadoop/sbin

[root@master sbin]# ./mr-jobhistory-daemon.sh start historyserver

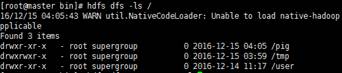
https://bd.cstor.cn/experiment/exp28.files/image001.jpg

（1）单词计数：

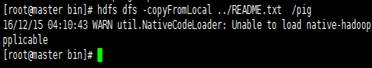
在HDFS文件系统创建一个pig目录，并从本地上传任意一个文件到目录中，为后面的Pig单词统计准备数据：

cd /usr/cstor/hadoop/bin

hdfs dfs -mkdir /pig



hdfs dfs -copyFromLocal ../README.txt /pig

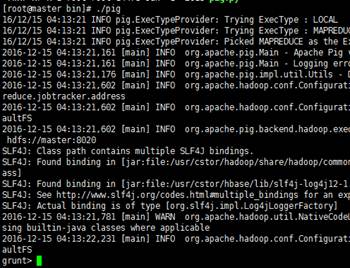




HDFS数据准备好后，再进入pig的bin目录执行下面的命令，进入pig编程界面：

[root@master cstor]# cd /usr/cstor/pig/bin

[root@master bin]# ./pig



代码实现如下：

#按行读HDFS文件

A = LOAD '/pig/README.txt' AS (line: chararray) ;

#将每行单词用tab、逗号，句号以及空格分隔单词

B = foreach A generate flatten(TOKENIZE(line,'\t ,.'))as word;

#按单词分组，将相同的单词归并到一起

C = group B by word;

#统计每个单词的个数

D = foreach C generate group, COUNT(B) as count;

#将结果显示出来

DUMP D;

（2）结构化数据查询统计：

在Pig节点的/root/data/28目录下新建文本文件pigdata.txt，

[root@master 28]# touch pigdata.txt

内容如下：

zhangsan,m,28

lisi,f,26

wangwu,m,20

将pigdata.txt上传到HDFS的/pig目录下：

cd /usr/cstor/hadoop/bin

[root@master bin]# hdfs dfs -copyFromLocal /root/data/28/pigdata.txt /pig

进入Pig编程界面：

cd /usr/cstor/pig/bin

[root@master bin]# ./pig

代码实现如下：

#按行读HDFS文件，并将数据按逗号分隔解析成姓名，性别，年龄

A = LOAD '/pig/pigdata.txt' USING PigStorage(',') AS(name:chararray, sex:chararray, age:int);

#打印A中数据

DUMP A;

#只查询name字段，并打印

B = FOREACH A GENERATE name;

DUMP B;

#查询所有人员数据，并将年龄加一

C = FOREACH A GENERATE name, sex, age+1;

DUMP C;

#查询年龄大于20的人员

D = FILTER A BY (age > 20);

DUMP D;

#统计数据总条数

E = GROUP A ALL ;

F = FOREACH E GENERATE COUNT (A);

DUMP F;

28.5 实验结果

Pig客户端输出结果如图28-1所示。



图28-1 Pig任务完成情况图

Pig客户端单词统计结果如图28-2所示。



图28-2 单词统计图

查询全部人员结果如图28-3所示。

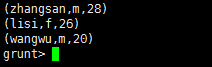


图28-3 查询全部人员图

查询全部人员姓名结果如图28-4所示。

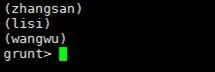


图28-4 查询全部人员姓名图

查询所有人信息，并且年龄加一，结果如图28-5所示。

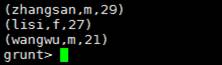


图28-5 查询年龄加一图

查询年龄大于20的人员，结果如图28-6所示。

https://bd.cstor.cn/experiment/exp28.files/image011.jpg

图28-6 查询年龄大于20的人员图

统计数据总条数，结果如图28-7所示。

https://bd.cstor.cn/experiment/exp28.files/image012.jpg

图28-7 统计数据总条数图

实验二十九 Redis部署与简单使用

29.1 实验目的

1． 熟悉了解CentOS系统；

2． 学习从C++源代码编译成可执行文件并安装运行；

3． 学习安装Redis；

4． 学习配置Redis；

5． 学会简单使用Redis。

29.2 实验要求

在client机上，安装配置并简单使用Redis。

29.3 实验原理

29.3.1 CentOS 简介

全称为Community Enterprise Operating System（社区企业操作系统），是Linux发行版之一。它是来自于Red Hat Enterprise Linux依照开放源代码规定释出的源代码所编译而成。由于出自同样的源代码，因此有些要求高度稳定性的服务器以CentOS替代商业版的Red Hat Enterprise Linux（RHEL）使用。两者的不同，在于CentOS并不包含封闭源代码软件。最新的CentOS版本是7.2，内核版本是3.10.0。

29.3.2 CentOS与RHEL关系

CentOS与RHEL的关系： RHEL 在发行的时候，有两种方式。一种是二进制的发行方式，另外一种是源代码的发行方式。无论是哪一种发行方式，你都可以免费获得（例如从网上下载），并再次发布。但如果你使用了他们的在线升级（包括补丁）或咨询服务，就必须要付费。RHEL 一直都提供源代码的发行方式，CentOS 就是将 RHEL 发行的源代码重新编译一次，形成一个可使用的二进制版本。由于 LINUX 的源代码是 GNU，所以从获得 RHEL 的源代码到编译成新的二进制，都是合法。只是 red hat 是商标，所以必须在新的发行版里将red hat 的商标去掉。red hat对这种发行版的态度是："我们其实并不反对这种发行版，真正向我们付费的用户，他们重视的并不是系统本身，而是我们所提供的商业服务。" 所以，CentOS 可以得到 RHEL 的所有功能，甚至是更好的软件。但 CentOS 并不向用户提供商业支持，当然也不负上任何商业责任。如果你要将你的 RHEL 转到 CentOS 上，因为你不希望为 RHEL 升级而付费。当然，你必须有丰富 linux 使用经验，因此 RHEL 的商业技术支持对你来说并不重要。但如果你是单纯的业务型企业，那么还是建议你选购 RHEL 软件并购买相应服务。这样可以节省你的 IT 管理费用，并可得到专业服务。一句话，选用 CentOS 还是 RHEL，取决于你所在公司是否拥有相应的技术力量。

29.3.3 make简介

Linux 下 make 命令是系统管理员和程序员用的最频繁的命令之一。管理员用它通过命令行来编译和安装很多开源的工具，程序员用它来管理他们大型复杂的项目编译问题。make命令通常和Makefile一起使用。

在开发一个系统时，一般是将一个系统分成几个模块，这样做提高了系统的可维护性，但由于各个模块间不可避免存在关联，所以当一个模块改动后，其他模块也许会有所更新，当然对小系统来说，手工编译连接是没问题，但是如果是一个大系统，存在很多个模块，那么手工编译的方法就不适用了。为此，在Linux系统中，专门提供了一个make命令来自动维护目标文件，与手工编译和连接相比，make命令的优点在于他只更新修改过的文件（在Linux中，一个文件被创建或更新后有一个最后修改时间，make命令就是通过这个最后修改时间来判断此文件是否被修改），而对没修改的文件则置之不理，并且make命令不会漏掉一个需要更新的文件。

文件和文件间或模块或模块间有可能存在倚赖关系，make命令也是依据这种依赖关系来进行维护的，所以我们有必要了解什么是依赖关系;make命令当然不会自己知道这些依赖关系，而需要程序员将这些依赖关系写入一个叫makefile的文件中。Makefile文件中包含着一些目标，通常目标就是文件名，对每一个目标，提供了实现这个目标的一组命令以及和这个目标有依赖关系的其他目标或文件名，以下是一个简单的Makefile的简单例子：

　　#一个简单的Makefile

　　prog:prog1.o prog2.o

　　 gcc prog1.o prog2.o -o prog

　　prog1.o:prog1.c lib.h

　　 gcc -c -I. -o prog1.o prog1.c

　　prog2.o:prog2.c

　　 gcc -c prog2.c

以上Mamefile中定义了三个目标：prog、prog1和prog2，冒号后是依赖文件列表;

对于第一个目标文件prog来说，他有两个依赖文件：prog1.o和prog2.o，任何一个依赖文件更新，prog也要随之更新，命令gcc prog1.o prog2.o -o prog是生成prog的命令。make检查目标是否需要更新时采用递归的方法，递归从底层向上对过时目标进行更新，只有当一个目标所依赖的所有目标都为最新时，这个目标才会被更新。以上面的Makefile为例，我们修改了prog2.c，执行make时，由于目标prog依赖prog1.o和prog2.o，所以要先检查prog1.o和prog2.o是否过时，目标prog1.o依赖prog1.c和lib.h，由于我们并没修改这两个文件，所以他们都没有过期，接下来再检查目标prog2.o，他依赖prog2.c，由于我们修改了prog2.c，所以prog2.c比目标文件prog2.o要新，即prog2.o过期，而导致了依赖prog2.o的所有目标都过时;这样make会先更新prog2.o再更新prog。

29.3.4 Redis简介

Redis是一个开源的使用ANSI C语言编写、支持网络、可基于内存亦可持久化的日志型、Key-Value数据库，并提供多种语言的API。从2010年3月15日起，Redis的开发工作由VMware主持。从2013年5月开始，Redis的开发由Pivotal赞助。

redis是一个key-value存储系统。和Memcached类似，它支持存储的value类型相对更多，包括string(字符串)、list(链表)、set(集合)、zset(sorted set --有序集合)和hash（哈希类型）。这些数据类型都支持push/pop、add/remove及取交集并集和差集及更丰富的操作，而且这些操作都是原子性的。在此基础上，redis支持各种不同方式的排序。与memcached一样，为了保证效率，数据都是缓存在内存中。区别的是redis会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了master-slave(主从)同步。

Redis 是一个高性能的key-value数据库。 redis的出现，很大程度补偿了memcached这类key/value存储的不足，在部 分场合可以对关系数据库起到很好的补充作用。它提供了Java，C/C++，C#，PHP，JavaScript，Perl，Object-C，Python，Ruby，Erlang等客户端，使用很方便。

Redis支持主从同步。数据可以从主服务器向任意数量的从服务器上同步，从服务器可以是关联其他从服务器的主服务器。这使得Redis可执行单层树复制。存盘可以有意无意的对数据进行写操作。由于完全实现了发布/订阅机制，使得从数据库在任何地方同步树时，可订阅一个频道并接收主服务器完整的消息发布记录。同步对读取操作的可扩展性和数据冗余很有帮助。

29.4 实验步骤

本实验包括安装、配置与使用三部分，下面按此三部分依次讲述。

29.4.1 安装配置启动

首先，登录client机，在该机上，依次输入下述命令安装即可，注意“make”与“make install”命令执行时会有大量输出（故此处并未截图），请读者确保这两个命令都在“/usr/cstor/redis”目录下执行即可，其他暂不必理会。

[root@client ~]# cd /usr/cstor/redis/

[root@client redis]# make

[root@client redis]# make install

接着，请读者使用“vim”，编辑“/usr/cstor/redis/redis.conf”文件，定位到下述内容所在行：

bind 127.0.0.1

将上述的“127.0.0.1”换成本机IP，笔者此处为“10.1.1.76”（图29-1）。

bind 172.17.0.64

最后，使用“redis-server”命令，指定上述配置文件，启动Redis，操作如图29-1所示：

[root@client ~]# redis-server /usr/cstor/redis/redis.conf &

https://bd.cstor.cn/experiment/exp29.files/image001.jpg

图29-1 已后台服务方式启动redis数据库

29.4.2 使用Redis

连接本地redis

[root@client redis]$ src/redis-cli

连接远程redis

[root@master redis]$ src/redis-cli -h client

向redis里写入数据

[root@client redis]$ src/redis-cli -h 172.17.0.15

172.17.0.15:6379> set chengshi sh,bj,sz,nj,hf

172.17.0.15:6379> get chengshi

实验三十 MapReduce与Spark读写Redis

30.1 实验目的

1． 会使用MapReduce访问Redis数据；

2． 会使用Spark访问Redis数据。

30.2 实验要求

1． 在client机上，使用MapReduce代码读取Redis数据；

2． 在client机上，使用Spark代码读取Redis数据。

30.3 实验原理

假定现有一个大为1000G的大表big.txt和一个大小为10G的小表small.txt，请基于MapReduce思想编程实现判断小表中单词在大表中出现次数。也即所谓的“扫描大表、加载小表”。

为解决上述问题，可开启10个Map。这样，每个Map只需处理总量的1/10，将大大加快处理。而在单独Map内，由于10G的small.txt依旧非常巨大，显然不适宜HashSet加载到内存等措施，此时可借助第三方存储介质（如Redis），在Map阶段先加载部分数据，然后再加载部分数据，甚至，可直接将结果写入Redis如图30-1所示。

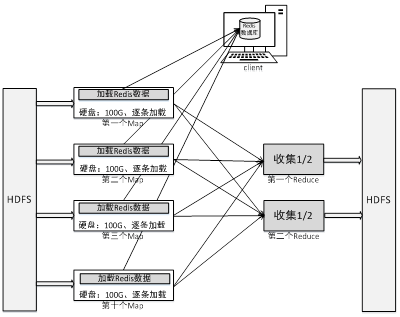


图30-1 Map阶段访问Redis

30.4 实验步骤

主要包括MapReduce读取Redis代码和Spark读取Redis数据代码。

30.4.1 MapReduce读取Redis

首先是准备数据阶段，即准备好big.txt并将其上传至HDFS，准备好待查城市名并将其导入Redis。接着为编程阶段，即编写MapReduce程序，在此程序Map阶段，取出Redis里待查城市，顺序扫描数据块里数据，是该城市则输出，不是则不做任何操作。最后，在集群上执行该程序。

30.4.1.1 准备HDFS数据

首先，登录client机或者master机器，确认该机上存在“/root/data/30/big.txt”(如果不存在请自己手动添加这样的文本数据)，如图30-2所示。

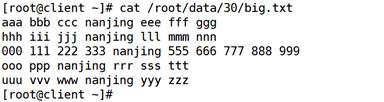


图30-2 确认本地文件big.txt

接着，登录到client机或者master机器上，查看HDFS里是否已存在目录“/user/root/redis/in”，若不存在，使用下述命令新建该目录。

[root@client ~]# /usr/cstor/hadoop/bin/hdfs dfs -mkdir -p /user/root/redis/in

最后，使用下述命令将client机本地文件“/root/data/30/big.txt”上传至HDFS的“/user/root/redis/in”目录：

[root@client ~]# /usr/cstor/hadoop/bin/hdfs dfs -put /root/data/30/big.txt /user/root/redis/in

最后，请进入HDFS Web页面，确认HDFS上文件与内容，如图30-3所示。

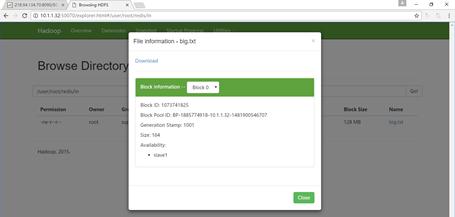


图30-3 确认HDFS上big.txt文件位置与内容

也可以使用命令行查看文件：

[root@client ~]# /usr/cstor/hadoop/bin/hdfs dfs -ls /user/root/redis/in

30.4.1.2 准备Redis数据

首先，参考下述命令登录到redis数据库：

进入redis的src目录：cd /usr/cstor/redis/src

然后登陆redis

[root@client ~]# redis-cli -h 10.1.1.36（这里是运行redis服务器的主机的ip）

接着，向Redis数据库里写入<city,nanjing>，参考命令如下：

10.1.1.36:6379> set city nanjing

10.1.1.36:6379> get city

30.4.1.3 编程MapReduce程序

首先，打开Eclipse，依次点击“FileNewOther…Map/Reduce Project”，在弹出的“New MapReduce Project Wizard”对话框中，“Project name:”一栏填写项目名“SharedLargeMemory”，然后直接点击该对话框的“Finish”按钮。

接着，新建LargeMemory类并指定包名（代码中为cn.cstor.redis），在LargeMemory.java文件中，依次写入如下代码：

package cn.cstor.redis;

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.Reducer;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import redis.clients.jedis.Jedis;

public class LargeMemory {

public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable> {

private final static IntWritable one = new IntWritable(1);

Jedis jedis = null;

protected void setup(Context context) throws IOException, InterruptedException {

jedis = new Jedis(context.getConfiguration().get("redisIP"));

System.out.println("setup ok \*^\_^\* ");

}

public void map(Object key, Text value, Context context) throws IOException, InterruptedException {

String[] values = value.toString().split(" ");

for (int i = 0; i < values.length; i++) {

if (jedis.get("city").equals(values[i])) {

context.write(new Text(values[i]), one);

}

}

}

}

public static class IntSumReducer extends Reducer<Text, IntWritable, Text, IntWritable> {

private IntWritable result = new IntWritable();

public void reduce(Text key, Iterable<IntWritable> values, Context context)

throws IOException, InterruptedException {

int sum = 0;

for (IntWritable val : values) {

sum += val.get();

}

result.set(sum);

context.write(key, result);

}

}

public static void main(String[] args) throws Exception {

Configuration conf = new Configuration();

conf.set("redisIP", args[0]);

Job job = Job.getInstance(conf, "RedisDemo");

job.setJarByClass(LargeMemory.class);

job.setMapperClass(TokenizerMapper.class);

job.setReducerClass(IntSumReducer.class);

job.setMapOutputKeyClass(Text.class);

job.setMapOutputValueClass(IntWritable.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(IntWritable.class);

FileInputFormat.addInputPath(job, new Path(args[1]));

FileOutputFormat.setOutputPath(job, new Path(args[2]));

System.exit(job.waitForCompletion(true) ? 0 : 1);

}

}

由于该程序依赖Redis，故，还需添加Redis客户端Jar包。实际操作时，选中该项目，“NewFolder”，在弹出的对话框中填写“lib”，接着将“jedis-2.1.0.jar”(这个jedis-2.1.0.jar包如果实验环境下找不到可以从网络上下载也是同样可以使用的) 复制到该文件夹（“lib”）下，最后，选中“jedis-2.1.0.jar”，依次点击“Build PathAdd to Build Path”完成添加依赖包，如图30-4所示。

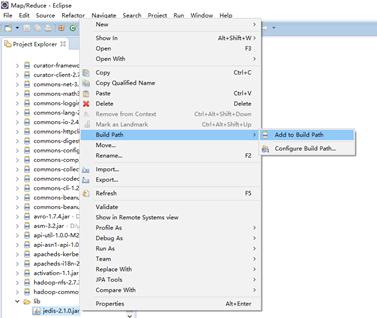


图30-4 将Jar包添加到Build Path

至此，已完成代码开发，图30-5为本项目结构图，请读者对照该图，分析项目结构图中各模块。

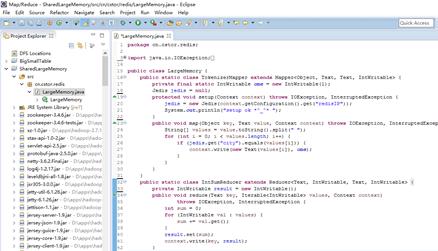


图30-5 本项目集成开发界面

30.4.1.4 打包该程序

待代码编写结束，选中该项目，依次点击“ExportJavaJAR file”，弹出对话框，在图30-6中填写打包位置，接着Finish即可。笔者此处打包时包名及其位置为“C:\Users\allen\Desktop\SharedLargeMemory.jar”。

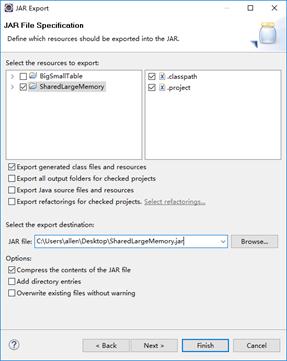


图30-6 项目打包

30.4.1.5 执行代码

首先，使用xftp或者一体机提供的工具将“C:\Users\allen\Desktop\SharedLargeMemory.jar”上传至client机，此处上传至“/root/SharedLargeMemory.jar”。

接着，登录client机上，使用下述命令提交SharedLargeMemory.jar任务。

[root@client ~]# /usr/cstor/hadoop/bin/hadoop jar /root/SharedLargeMemory.jar

cn.cstor.redis.LargeMemory 10.1.1.36 /user/root/redis/in/big.txt /user/root/redis/SLMResult

30.4.2 Spark读取Redis

和MapReduce相比，使用Spark读写Redis则简单的多，首先登录Redis准备数据；接着，在启动Spark时指定Redis包；最后，在Spark交互式执行界面中直接编写代码访问Redis即可。具体操作如下：

30.4.2.1 准备Redis数据

登录Redis，向redis数据库添加数据

[root@client ~]$ redis-cli -h client

172.17.0.15:6379> SET chengshi sh,bj,sz,nj,hf

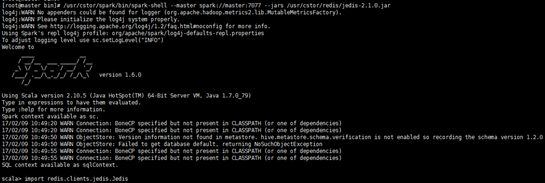
172.17.0.15:6379> get chengshi

30.4.2.2 启动SparkShell

首先启动Spark集群，再启动SparkShell，由于Spark访问Redis时需要Redis客户端jedis，故此处启动Spark时需指定jedis包(如果redis的jar包不存在，可以手动放到/usr/cstor/redis/这个目录下)，启动命令如下：

[root@client ~]$/usr/cstor/spark/bin/spark-shell --master spark://master:7077 --jars /usr/cstor/redis/jedis-2.1.0.jar

启动成功如下：



30.4.2.3 编写访问代码

在交互代码中连接Redis(注意redis的IP地址修改成自己的)，读者可按下述命令操作：

scala> import redis.clients.jedis.Jedis

import redis.clients.jedis.Jedis

scala> var jd=new Jedis("172.17.0.15",6379)

jd: redis.clients.jedis.Jedis = redis.clients.jedis.Jedis@7dae9ff4

scala> var str=jd.get("chengshi")

str: String = sh,bj,sz,nj,hf

scala> var strList=str.split(",")

strList: Array[String] = Array(sh, bj, sz, nj, hf)

scala> val a = sc.parallelize(strList, 3)

a: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[0] at parallelize at <console>:28

scala> val b = a.keyBy(\_.length)

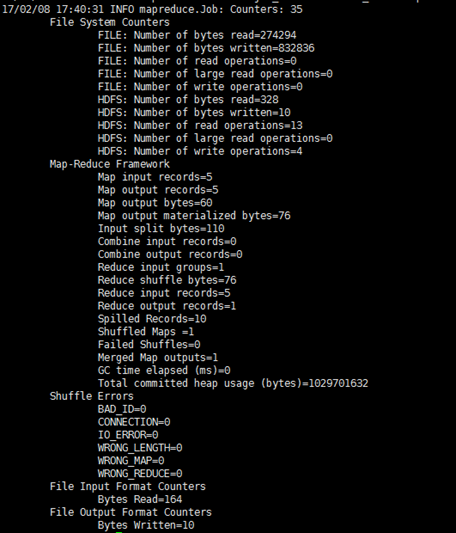
b: org.apache.spark.rdd.RDD[(Int, String)] = MapPartitionsRDD[1] at keyBy at <console>:30

scala> b.collect

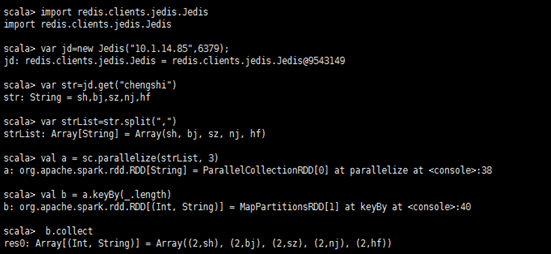
res0: Array[(Int, String)] = Array((2,sh), (2,bj), (2,sz), (2,nj), (2,hf))

30.5 实验结果

30.5.1 MapReduce读取Redis实验



30.5.2 Spark读取Redis实验



实验三十一 MongoDB实验：读写MongoDB

31.1 实验目的

1． 了解NoSQL 数据库的原理；

2． 理解NoSQL数据库的结构；

3． 比较MongoDB和Hbase的区别；

4． 能对MongoDB的存储格式有一定的了解；

5． 能正确的使用MongoDB并能进行简单使用。

31.2 实验要求

1． 正确的搭建MongoDB数据库环境；

2． 能正常启动MongoDB的服务和服务的连接；

3． 能在MongoDB的的shell中进行一些简单的使用。

31.3 实验原理

MongoDB是一个基于分布式文件存储的数据库。由C++语言编写。旨在为WEB应用提供可扩展的高性能数据存储解决方案。

特点：高性能、易部署、易使用，存储数据非常方便。

主要功能特性有：

1． 面向集合存储，易存储对象类型的数据；

2． 模式自由；

3． 支持动态查询；

4． 支持完全索引，包含内部对象；

5． 支持查询；

6． 支持复制和故障恢复；

7． 使用高效的二进制数据存储，包括大型对象（如视频等）；

8． 自动处理碎片，以支持云计算层次的扩展性；

9． 支持Ruby，Python，Java，C++，PHP等多种语言；

10． 文件存储格式为BSON（一种JSON的扩展）；

11． 可通过网络访问。

所谓“面向集合”（Collenction-Oriented），意思是数据被分组存储在数据集中，被称为一个集合（Collenction)。每个集合在数据库中都有一个唯一的标识名，并且可以包含无限数目的文档。集合的概念类似关系型数据库（RDBMS）里的表（table），不同的是它不需要定义任何模式（schema)。

模式自由（schema-free)，意味着对于存储在mongodb数据库中的文件，我们不需要知道它的任何结构定义。如果需要的话，你完全可以把不同结构的文件存储在同一个数据库里。

存储在集合中的文档，被存储为键-值对的形式。键用于唯一标识一个文档，为字符串类型，而值则可以是各种复杂的文件类型。我们称这种存储形式为BSON（Binary JSON）。

31.4 实验步骤

本实验总体分为几个步骤：

1． 启动MongoDB服务；

2． 连接MongoDB服务；

3． 启动MongoDB的shell，执行一些简单命令。

详细如下：

31.4.1 启动MongoDB

首先在MongoDB的安装目录下建立一个数据目录。然后进入MongoDB安装目录下的bin目录，启动MongoDB服务。

[root@master ~]# cd /usr/cstor/mongodb/

[root@master mongodb]# mkdir data

[root@master mongodb]# bin/mongod --dbpath ./data &

31.4.2 连接使用MongoDB

使用命令连接到MongoDB服务。

[root@master mongodb]# bin/mongo

31.4.3 连接启动MongoDB的shell，执行一些简单命令。

连接到MongoDB之后，进入shell环境之后执行如下的简单操作，以此来熟悉MongoDB的操作。

31.4.3.1 创建一个Collection

如图31-1所示：

> db.createCollection("weather");

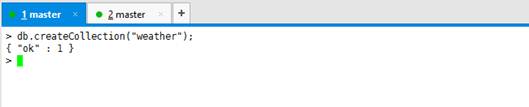


图31-1 创建collection

31.4.3.2 插入单条数据

如图31-2所示：

> db.weather.save({temp:31,location:"nanjing"});

https://bd.cstor.cn/experiment/exp31.files/image002.png

图31-2 插入单条数据

31.4.3.3 插入多条数据

如图31-3所示：

> for(var i = 0;i < 30;i++) { db.weather.save({tep:i,location:"nanjing"}) };

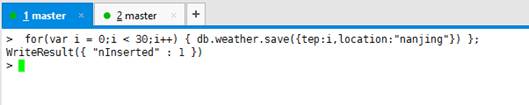


图31-3 插入多条数据

31.4.3.4 简单遍历

如图31-4所示：

> db.weather.find();

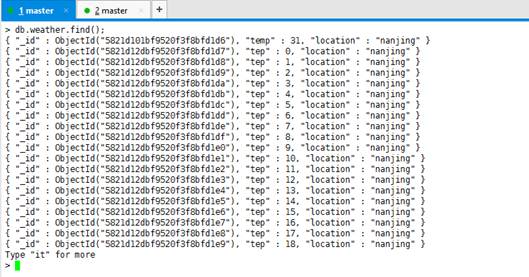


图31-4 简单遍历

31.4.3.5 复杂遍历

如图31-5所示：

> var c=db.weather.find();

>while(c.hasNext()){printjson(c.next())};

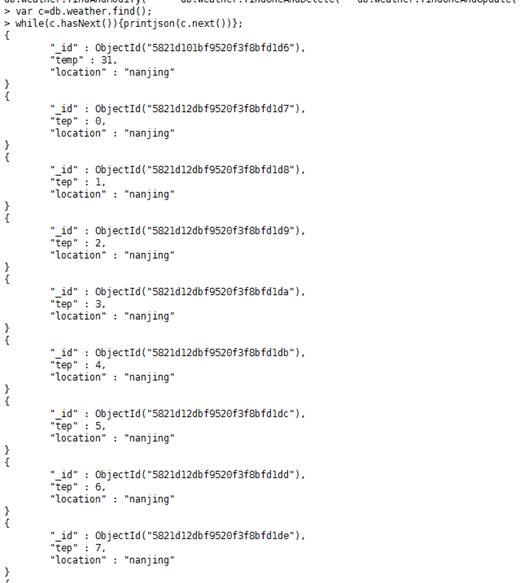


图31-5 复杂遍历

31.4.3.6 查找数据

如图31-6所示：

> db.weather.find({location:"najing"})

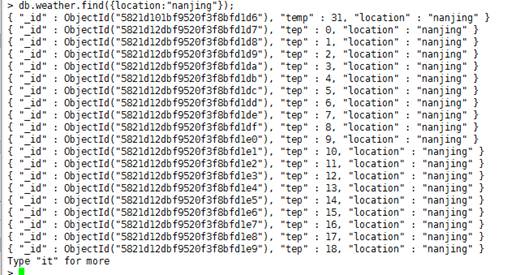


图31-6 查找数据

31.4.3.7 更新数据

如图31-7所示：

>db.weather.update({location:"nanjing"},{$set:{location:"shanghai"}},false,true);

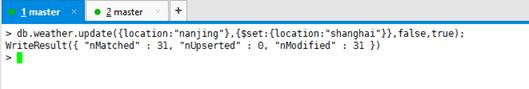


图31-7 更新数据

31.4.3.8 删除数据

如图31-8所示：

> db.weather.remove({location:"nanjing"})

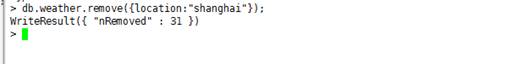


图31-8 删除数据

31.4.3.9 删除Collection

如图31-9所示：

>db.weather.drop() ;

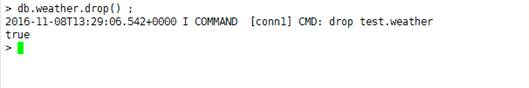


图31-9 删除collection

实验三十二 LevelDB实验：读写LevelDB

32.1 实验目的

1． 了解LevelDB的使用场景；

2． 理解LevelDB数据存储结构；

3． 比较LevelDB和redis的区别；

4． 能对LevelDB的整体架构有一定的了解；

5． 能正确的使用LevelDB并能进行简单使用。

32.2 实验要求

本实验要求同学能够使用C++语言完成对LevelDB库完成以下操作：

1． 连接到LevelDB数据库；

2． 写入数据；

3． 读取数据；

4． 删除数据。

32.3 实验原理

LevelDB是Google开源的持久化KV单机数据库，具有很高的随机写，顺序读/写性能，但是随机读的性能很一般，也就是说，LevelDB很适合应用在查询较少，而写很多的场景。LevelDB应用了[LSM](http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.44.2782&rep=rep1&type=pdf) (Log Structured Merge) 策略，lsm\_tree对索引变更进行延迟及批量处理，并通过一种类似于归并排序的方式高效地将更新迁移到磁盘，降低索引插入开销。

特点：

1． key和value都是任意长度的字节数组；

2． entry（即一条K-V记录）默认是按照key的字典顺序存储的，当然开发者也可以重载这个排序函数；

3． 提供的基本操作接口：Put()、Delete()、Get()、Batch()；

4． 支持批量操作以原子操作进行；

5． 可以创建数据全景的snapshot(快照)，并允许在快照中查找数据；

6． 可以通过前向（或后向）迭代器遍历数据（迭代器会隐含的创建一个snapshot）；

7． 自动使用Snappy压缩数据；

8． 可移植性。

限制：

1． 非关系型数据模型（NoSQL），不支持sql语句，也不支持索引；

2． 一次只允许一个进程访问一个特定的数据库；

3． 没有内置的C/S架构，但开发者可以使用LevelDB库自己封装一个server。

整体架构：LevelDB作为存储系统，数据记录的存储介质包括内存以及磁盘文件，如果像上面说的，当LevelDB运行了一段时间，此时我们给LevelDB进行透视拍照，那么您会看到如图32-1所示。

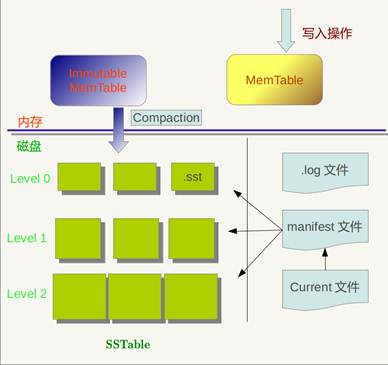


图32-1 LevelDB结构

从图中可以看出，构成LevelDB静态结构的包括六个主要部分：内存中的MemTable和Immutable MemTable以及磁盘上的几种主要文件：Current文件，Manifest文件，log文件以及SSTable文件。当然，LevelDB除了这六个主要部分还有一些辅助的文件，但是以上六个文件和数据结构是LevelDB的主体构成元素。

LevelDB的Log文件和Memtable与Bigtable论文中介绍的是一致的，当应用写入一条Key:Value记录的时候，LevelDB会先往log文件里写入，成功后将记录插进Memtable中，这样基本就算完成了写入操作，因为一次写入操作只涉及一次磁盘顺序写和一次内存写入，所以这是为何说LevelDB写入速度极快的主要原因。

Log文件在系统中的作用主要是用于系统崩溃恢复而不丢失数据，假如没有Log文件，因为写入的记录刚开始是保存在内存中的，此时如果系统崩溃，内存中的数据还没有来得及Dump到磁盘，所以会丢失数据（Redis就存在这个问题）。为了避免这种情况，LevelDB在写入内存前先将操作记录到Log文件中，然后再记入内存中，这样即使系统崩溃，也可以从Log文件中恢复内存中的Memtable，不会造成数据的丢失。

当Memtable插入的数据占用内存到了一个界限后，需要将内存的记录导出到外存文件中，LevelDB会生成新的Log文件和Memtable，原先的Memtable就成为Immutable Memtable，顾名思义，就是说这个Memtable的内容是不可更改的，只能读不能写入或者删除。新到来的数据被记入新的Log文件和Memtable，LevelDB后台调度会将Immutable Memtable的数据导出到磁盘，形成一个新的SSTable文件。SSTable就是由内存中的数据不断导出并进行Compaction操作后形成的，而且SSTable的所有文件是一种层级结构，第一层为Level 0，第二层为Level 1，依次类推，层级逐渐增高，这也是为何称之为LevelDB的原因。

SSTable中的文件是Key有序的，就是说在文件中小key记录排在大Key记录之前，各个Level的SSTable都是如此，但是这里需要注意的一点是：Level 0的SSTable文件（后缀为.sst）和其它Level的文件相比有特殊性：这个层级内的.sst文件，两个文件可能存在key重叠，比如有两个level 0的sst文件，文件A和文件B，文件A的key范围是：{bar, car}，文件B的Key范围是{blue,samecity}，那么很可能两个文件都存在key=”blood”的记录。对于其它Level的SSTable文件来说，则不会出现同一层级内.sst文件的key重叠现象，就是说Level L中任意两个.sst文件，那么可以保证它们的key值是不会重叠的。这点需要特别注意，后面您会看到很多操作的差异都是由于这个原因造成的。

SSTable中的某个文件属于特定层级，而且其存储的记录是key有序的，那么必然有文件中的最小key和最大key，这是非常重要的信息，LevelDB应该记下这些信息。Manifest就是干这个的，它记载了SSTable各个文件的管理信息，比如属于哪个Level，文件名称叫啥，最小key和最大key各自是多少。如图32-2是Manifest所存储内容的示意：

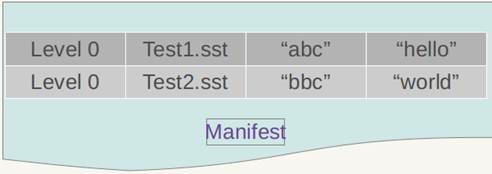


图32-2 Manifest存储示意图

图中只显示了两个文件（manifest会记载所有SSTable文件的这些信息），即Level 0的test.sst1和test.sst2文件，同时记载了这些文件各自对应的key范围，比如test.sst1的key范围是“an”到 “banana”，而文件test.sst2的key范围是“baby”到“samecity”，可以看出两者的key范围是有重叠的。

Current文件是干什么的呢？这个文件的内容只有一个信息，就是记载当前的manifest文件名。因为在LevleDb的运行过程中，随着Compaction的进行，SSTable文件会发生变化，会有新的文件产生，老的文件被废弃，Manifest也会跟着反映这种变化，此时往往会新生成Manifest文件来记载这种变化，而Current则用来指出哪个Manifest文件才是我们关心的那个Manifest文件。

以上介绍的内容就构成了LevelDB的整体静态结构。

32.4 实验步骤

在client机上操作：首先进入LevelDB目录。

[root@client ~]# cd /usr/cstor/leveldb

[root@client leveldb]#

然后创建一个code的文件夹作为代码编写的目录。

[root@client leveldb]# mkdir code

进入code目录。

[root@client leveldb# cd code

[root@client code]#

利用vim编写leveldb.cpp代码（请参考32.4.6小节），

编译leveldb.cpp，编译命令如下：

[root@client code]# g++ -o leveldb leveldb.cpp ../out-static/libleveldb.a -lpthread -I../include

32.4.1 使用C++代码建立数据库连接

核心C++代码如下：

// 打开数据库连接

leveldb::Status status = leveldb::DB::Open(options,"./test\_level\_db", &db);

assert(status.ok());

string key = "weather";

string value = "clearday";

32.4.2 写入数据

写入数据的核心C++代码如下：

status = db->Put(leveldb::WriteOptions(), key, value);

assert(status.ok());

32.4.3 读取数据

读取数据的核心C++代码如下：

status = db->Get(leveldb::ReadOptions(), key, &value);

assert(status.ok());

cout<<"value :"<<value<<endl;

32.4.4 删除数据

删除数据的核心C++代码如下：

status = db->Delete(leveldb::WriteOptions(), key);

assert(status.ok());

status = db->Get(leveldb::ReadOptions(),key, &value);

if(!status.ok()) {

cerr<<key<<" "<<status.ToString()<<endl;

} else {

cout<<key<<"==="<<value<<endl;

}

32.4.5 关闭连接

关闭连接的核心C++代码如下：

delete db;

32.4.6 完整的代码

#include <iostream>

#include <string>

#include <assert.h>

#include "leveldb/db.h"

using namespace std;

int main(void)

{

leveldb::DB \*db;

leveldb::Options options;

options.create\_if\_missing = true;

// 打开数据库连接

leveldb::Status status = leveldb::DB::Open(options,"./test\_level\_db", &db);

assert(status.ok());

string key = "weather";

string value = "clearday";

// 写入数据

status = db->Put(leveldb::WriteOptions(), key, value);

assert(status.ok());

// 读取数据

status = db->Get(leveldb::ReadOptions(), key, &value);

assert(status.ok());

cout<<"value :"<<value<<endl;

// 删除数据

status = db->Delete(leveldb::WriteOptions(), key);

assert(status.ok());

status = db->Get(leveldb::ReadOptions(),key, &value);

if(!status.ok()) {

cerr<<key<<" "<<status.ToString()<<endl;

} else {

cout<<key<<"==="<<value<<endl;

}

// 关闭连接

delete db;

return 0;

}

32.5 实验结果

输入 ./leveldb 运行程序查看结果。

[root@client code]# ./leveldb

如图32-3所示：

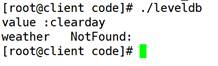


图32-3 leveldb结果

用ls命令查看test\_level\_db目录如下。如图32-4所示。

[root@client code]# ls test\_level\_db

如图32-4所示：

https://bd.cstor.cn/experiment/exp32.files/image004.jpg

图32-4 ls命令结果

实验三十三 Mahout实验：K-means

33.1 实验目的

1． 了解Mahout是什么；

2． 了解Mahout能够做什么；

3． 学会启动Mahout；

4． 能够通过提交MapReduce程序进行K-means实验。

33.2 实验要求

要求实验结束时，每位学生能够在Hadoop集群上利用Mahout提交K-means程序，并得出实验正确结果。

33.3 实验原理

33.3.1 Mahout简介

Apache Mahout 是 Apache Software Foundation (ASF) 开发的一个全新的开源项目，其主要目标是创建一些可伸缩的机器学习算法，供开发人员在 Apache 在许可下免费使用。该项目已经发展到了它的第二个年头，目前只有一个公共发行版。Mahout 包含许多实现，包括集群、分类、CP 和进化程序。此外，通过使用 Apache Hadoop 库，Mahout 可以有效地扩展到云中。

33.3.2 Mahout发展

Mahout 项目是由 Apache Lucene（开源搜索）社区中对机器学习感兴趣的一些成员发起的，他们希望建立一个可靠、文档翔实、可伸缩的项目，在其中实现一些常见的用于集群和分类的机器学习算法。该社区最初基于 Ng et al. 的文章 “Map-Reduce for Machine Learning on Multicore”，但此后在发展中又并入了更多广泛的机器学习方法。Mahout 的目标还包括：

（1）建立一个用户和贡献者社区，使代码不必依赖于特定贡献者的参与或任何特定公司和大学的资金。

（2）专注于实际用例，这与高新技术研究及未经验证的技巧相反。

（3）提供高质量文章和示例。

33.3.3 Mahout特性

虽然在开源领域中相对较为年轻，但 Mahout 已经提供了大量功能，特别是在集群和 CF 方面。Mahout 的主要特性包括：

（1）Taste CF。Taste 是 Sean Owen 在 SourceForge 上发起的一个针对 CF 的开源项目，并在 2008 年被赠予 Mahout。

（2）一些支持 Map-Reduce 的集群实现包括 K-means、模糊 K-means、Canopy、Dirichlet 和 Mean-Shift。

（3）Distributed Naive Bayes 和 Complementary Naive Bayes 分类实现。

（4）针对进化编程的分布式适用性功能。

（5）Matrix 和矢量库。

本实验主要介绍K-means。

33.3.4 K-means算法概要

K-means 算法是一种计算数据聚集的算法，如图33-1所示：

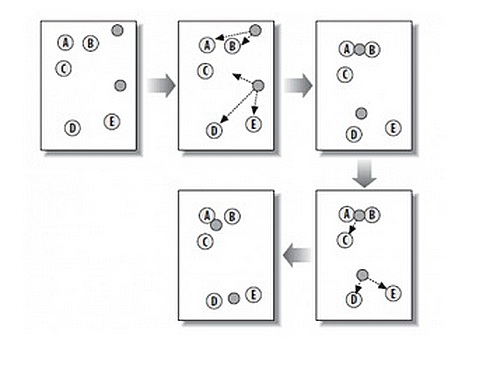


图33-1 K-means算法图示

从上图中，我们可以看到，A, B, C, D, E 是五个在图中点。而灰色的点是我们的种子点，也就是我们用来找点群的点。有两个种子点，所以K=2。

然后，K-means的算法如下：

（1）随机在图中取K（这里K=2）个种子点。

（2）然后对图中的所有点求到这K个种子点的距离，假如点Pi离种子点Si最近，那么Pi属于Si点群。（上图中，我们可以看到A,B属于上面的种子点，C,D,E属于下面中部的种子点）

（3）接下来，我们要移动种子点到属于他的“点群”的中心。（见图上的第三步）。

（4）然后重复第2）和第3）步，直到，种子点没有移动（我们可以看到图中的第四步上面的种子点聚合了A,B,C，下面的种子点聚合了D，E）。

33.3.5 K-means算法存在的问题

K-means 算法的特点——采用两阶段反复循环过程算法，结束的条件是不再有数据元素被重新分配：

指定聚类：

即指定数据到某一个聚类，使得它与这个聚类中心的距离比它到其它聚类中心的距离要近。

修改聚类中心：

优点：本算法确定的K 个划分到达平方误差最小。当聚类是密集的，且类与类之间区别明显时，效果较好。对于处理大数据集，这个算法是相对可伸缩和高效的，计算的复杂度为O(NKt)，其中N是数据对象的数目，t是迭代的次数。一般来说，K<<N，t<<N 。

33.3.6 K-means算法优点

K-means聚类算法的优点主要集中在:

（1）算法快速、简单；

（2）对大数据集有较高的效率并且是可伸缩性的；

（3）时间复杂度近于线性，而且适合挖掘大规模数据集。K-means聚类算法的时间复杂度是O(nkt) ,其中n代表数据集中对象的数量，t代表着算法迭代的次数，k代表着簇的数目。

33.3.7 K-means算法缺点

K-means 算法缺点

在 K-means 算法中 K 是事先给定的，这个 K 值的选定是非常难以估计的。很多时候，事先并不知道给定的数据集应该分成多少个类别才最合适。这也是 K-means 算法的一个不足。有的算法是通过类的自动合并和分裂，得到较为合理的类型数目 K，例如 ISODATA 算法。关于 K-means 算法中聚类数目K 值的确定在文献中，是根据方差分析理论，应用混合 F统计量来确定最佳分类数，并应用了模糊划分熵来验证最佳分类数的正确性。在文献中，使用了一种结合全协方差矩阵的 RPCL 算法，并逐步删除那些只包含少量训练数据的类。而文献中使用的是一种称为次胜者受罚的竞争学习规则，来自动决定类的适当数目。它的思想是：对每个输入而言，不仅竞争获胜单元的权值被修正以适应输入值，而且对次胜单元采用惩罚的方法使之远离输入值。

在 K-means 算法中，首先需要根据初始聚类中心来确定一个初始划分，然后对初始划分进行优化。这个初始聚类中心的选择对聚类结果有较大的影响，一旦初始值选择的不好，可能无法得到有效的聚类结果，这也成为 K-means算法的一个主要问题。对于该问题的解决，许多算法采用遗传算法（GA），例如文献 中采用遗传算法（GA）进行初始化，以内部聚类准则作为评价指标。

从 K-means 算法框架可以看出，该算法需要不断地进行样本分类调整，不断地计算调整后的新的聚类中心，因此当数据量非常大时，算法的时间开销是非常大的。所以需要对算法的时间复杂度进行分析、改进，提高算法应用范围。在文献中从该算法的时间复杂度进行分析考虑，通过一定的相似性准则来去掉聚类中心的侯选集。而在文献中，使用的 K-means 算法是对样本数据进行聚类，无论是初始点的选择还是一次迭代完成时对数据的调整，都是建立在随机选取的样本数据的基础之上，这样可以提高算法的收敛速度。

33.3.8 K-means算法应用

看到这里，你会说，K-means算法看来很简单，而且好像就是在玩坐标点，没什么真实用处。而且，这个算法缺陷很多，还不如人工呢。是的，前面的例子只是玩二维坐标点，的确没什么意思。但是你想一下下面的几个问题：

（1）如果不是二维的，是多维的，如5维的，那么，就只能用计算机来计算了。

（2）二维坐标点的X, Y 坐标，其实是一种向量，是一种数学抽象。现实世界中很多属性是可以抽象成向量的，比如，我们的年龄，我们的喜好，我们的商品，等等，能抽象成向量的目的就是可以让计算机知道某两个属性间的距离。如：我们认为，18岁的人离24岁的人的距离要比离12岁的距离要近等等。

只要能把现实世界的物体的属性抽象成向量，就可以用K-means算法来归类了。

33.4 实验步骤

33.4.1 添加临时JAVA\_HOME环境变量

添加临时JAVA\_HOME环境变量

[root@client hadoop]# export JAVA\_HOME=/usr/local/jdk1.7.0\_79

33.4.2 建立HDFS目录

在client机上操作：首先在HDFS上建立目录

[root@client hadoop]# cd /usr/cstor/hadoop/

[root@client hadoop]# bin/hadoop fs -mkdir -p /usr/cstor/hadoop/testdata

33.4.3 实验数据准备

然后将/root/data/33/文件夹下的synthetic\_control.data文件上传到HDFS上面刚刚新建好的目录下面。

[root@client hadoop]# bin/hadoop fs -put /root/data/33/synthetic\_control.data /usr/cstor/hadoop/testdata

33.4.4 提交Mahout的K-means程序

执行代码提交命令，提交Mahout的K-means程序。

[root@client hadoop]# bin/hadoop jar /usr/cstor/mahout/mahout-examples-0.9-job.jar \

> org.apache.mahout.clustering.syntheticcontrol.kmeans.Job

33.5 实验结果

在client上执行对HDFS上的文件夹/usr/cstor/hadoop/output内容查看的操作

[root@client hadoop]# bin/hadoop fs -ls /usr/cstor/hadoop/output/

屏幕上显示如图33-2所示：

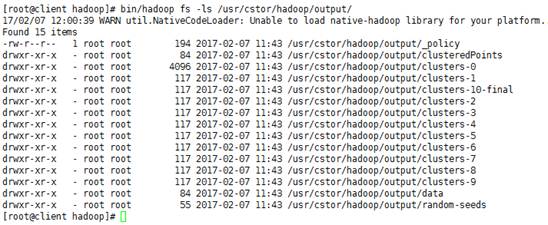


图33-2 查看文件列表

实验三十四 使用Spark实现K-Means

34.1 实验目的

1． 熟练使用spark-shell接口；

2． 了解K-Means算法原理；

3． 理解K-Means执行过程；

4． 配置Spark处理HDFS数据；

5． 使用Spark机器学习包中K-Means工具包处理HDFS中数据。

34.2 实验要求

使用Spark MLlib（机器学习库）中的K-Means工具包，对存储在HDFS上的数据集sample\_kmeans\_data.txt进行聚类。

34.3 实验原理

请参考实验三十三实验原理。

34.4 实验步骤

K-Means主要用于在对样本类别不了解情况下，对样本进行聚类，其本身无“训练、预测”这一说法，直接设置中心点数量或终止距离即可。下面将按照“准备数据集使用K-Means进行聚类”依次讲述。

34.4.1 添加临时JAVA\_HOME环境变量

首先，手工或者使用“一键搭建spark”功能构筑好spark环境。

登录slave1机，添加临时JAVA\_HOME环境变量

[root@slave1 ~]# export JAVA\_HOME=/usr/local/jdk1.7.0\_79

34.3.2 上传训练数据集

查看HDFS里是否已存在目录“/34/in”，若不存在，使用下述命令新建该目录。

[root@slave1 ~]# /usr/cstor/hadoop/bin/hdfs dfs -mkdir -p /34/in

https://bd.cstor.cn/experiment/exp34.files/image001.jpg

图34-1 创建目录

接着，使用下述命令将slave1机本地文件“/usr/cstor/spark/data/mllib/ kmeans\_data.txt”上传至HDFS的“/34/in”目录：

[root@slave1 ~]# /usr/cstor/hadoop/bin/hdfs dfs -put /usr/cstor/spark/data/mllib/kmeans\_data.txt /34/in

https://bd.cstor.cn/experiment/exp34.files/image002.jpg

图34-2 上传文件

最后，请确认HDFS上存在文件“/usr/cstor/spark/data/mllib/kmeans\_data.txt”。

https://bd.cstor.cn/experiment/exp34.files/image003.jpg

图34-3 确认文件是否上传成功

34.3.3 训练SVM模型

准备好输入文件后，下一步便是在Spark集群上执行K-Means程序（处理该数据集）。下面的处理代码参考自“/usr/cstor/spark/examples/src/main/scala/org/apache/ spark/examples/SparkKMeans.scala”，操作命令主要在client机上完成。

首先，在slave1机上，使用下述命令，进入spark-shell接口。

[root@slave1 ~]# /usr/cstor/spark/bin/spark-shell --master spark://master:7077

进入spark-shell命令行执行环境后，依次输入下述代码，即完成模型训练。

import breeze.linalg.{Vector, DenseVector, squaredDistance}

import org.apache.spark.{SparkConf, SparkContext}

import org.apache.spark.SparkContext.\_

def parseVector(line: String): Vector[Double] = {

DenseVector(line.split(' ').map(\_.toDouble))

}

def closestPoint(p: Vector[Double], centers: Array[Vector[Double]]): Int = {

var bestIndex = 0

var closest = Double.PositiveInfinity

for (i <- 0 until centers.length) {

val tempDist = squaredDistance(p, centers(i))

if (tempDist < closest) {

closest = tempDist

bestIndex = i

}

}

bestIndex

}

val lines = sc.textFile("/34/in/kmeans\_data.txt")

val data = lines.map(parseVector \_).cache()

val K = "2".toInt

val convergeDist = "0.1".toDouble

val kPoints = data.takeSample(withReplacement = false, K, 42).toArray

var tempDist = 1.0

while(tempDist > convergeDist) {

val closest = data.map (p => (closestPoint(p, kPoints), (p, 1)))

val pointStats = closest.reduceByKey{case ((p1, c1), (p2, c2)) => (p1 + p2, c1 + c2)}

val newPoints = pointStats.map {pair =>

(pair.\_1, pair.\_2.\_1 \* (1.0 / pair.\_2.\_2))}.collectAsMap()

tempDist = 0.0

for (i <- 0 until K) {

tempDist += squaredDistance(kPoints(i), newPoints(i))

}

for (newP <- newPoints) {

kPoints(newP.\_1) = newP.\_2

}

println("Finished iteration (delta = " + tempDist + ")")

}

println("Final centers:")

kPoints.foreach(println)

可以直接复制上述代码粘贴至在spark交换式执行器中执行。

34.5 实验结果

实验直接使用Spark自带的K-Means数据集与K-Means代码库，省去大量代码篇幅，读者可下载源码包，然后逐个跟进便可看到完整K-Means处理源码。

本实验完成使用K-Means算法对sample\_kmeans\_data.txt内所有样本进行聚类，代码最后一行便是在控制台上打印中心点，最终执行结果如图34-4所示。

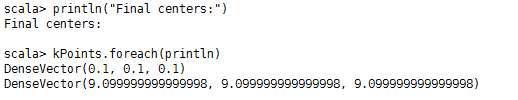


图34-4 实验结果

实验三十五 使用Spark实现SVM

35.1 实验目的

熟练使用Spark Shell接口；了解SVM（支持向量机）算法原理；理解SVM执行过程；配置Spark处理HDFS数据；使用Spark机器学习包中SVM工具包处理HDFS上数据。

35.2 实验要求

使用Spark MLlib（机器学习库）中的SVM工具包，训练存储在HDFS上的SVM训练数据集sample\_libsvm\_data.txt。

35.3 实验原理

35.3.1 SVM算法介绍

支持向量机SVM(Support Vector Machine)是一种分类算法，通过寻求结构化风险最小来提高学习机泛化能力，实现经验风险和置信范围的最小化，从而达到在统计样本量较少的情况下，亦能获得良好统计规律的目的。通俗来讲，它是一种二类分类模型，其基本模型定义为特征空间上的间隔最大的线性分类器，即支持向量机的学习策略便是间隔最大化，最终可转化为一个凸二次规划问题的求解。

35.3.2 SVM算法原理

（1）在n维空间中找到一个分类超平面，将空间上的点分类。如图35-1是线性分类的例子。



图35-1 SVM算法图示

（2）一般而言，一个点距离超平面的远近可以表示为分类预测的确信或准确程度。SVM就是要最大化这个间隔值。而在虚线上的点便叫做支持向量Supprot Verctor，如图35-2所示。

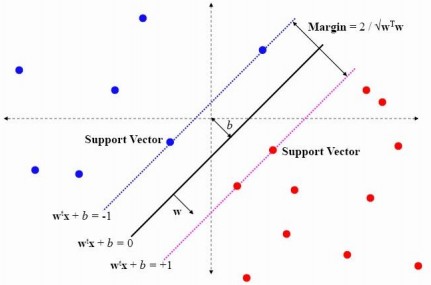


图35-2 支持向量

（3）实际中，我们会经常遇到线性不可分的样例，此时，我们的常用做法是把样例特征映射到高维空间中去，如图35-3所示。

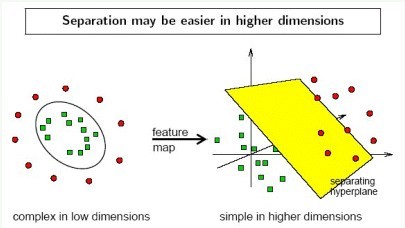


图35-3 高维映射

（3）线性不可分映射到高维空间，可能会导致维度大小高到可怕的(19维乃至无穷维的例子)，导致计算复杂。核函数的价值在于它虽然也是讲特征进行从低维到高维的转换，但核函数绝就绝在它事先在低维上进行计算，而将实质上的分类效果表现在了高维上，也就如上文所说的避免了直接在高维空间中的复杂计算。

（4）使用松弛变量处理数据噪音

35.4 实验步骤

当需要使用SVM算法对样本进行分类时，首先应基于已有样本，训练SVM模型，然后才可使用该模型对新样本进行预测。下面将按照“准备数据集训练SVM分类器使用SVM分类器”依次讲述。

35.4.1 添加临时JAVA\_HOME环境变量

首先，手工或者使用“一键搭建spark”功能构筑好spark环境。

登录slave1机，添加临时JAVA\_HOME环境变量

[root@slave1 ~]# export JAVA\_HOME=/usr/local/jdk1.7.0\_79

35.4.2 上传训练数据集

查看HDFS里是否已存在目录“/35/in”。若不存在，使用下述命令新建该目录。

[root@slave1 ~]# /usr/cstor/hadoop/bin/hdfs dfs -mkdir -p /35/in

接着，使用下述命令将slave1机本地文件“/root/data/35/sample\_libsvm\_data.txt”上传至HDFS的“/35/in”目录：

[root@slave1 ~]# /usr/cstor/hadoop/bin/hdfs dfs -put /root/data/35/sample\_libsvm\_data.txt /35/in

最后，请确认HDFS上已经存在文件sample\_libsvm\_data.txt。

https://bd.cstor.cn/experiment/exp35.files/image004.jpg

35.4.3 训练SVM模型

准备好输入文件后，下一步便是在Spark集群上执行SVM程序（处理该数据集）。下面的处理代码参考自“http://spark.apache.org/docs/latest/mllib-linear-methods.html”，操作命令主要在master机上完成。

首先，在slave1机上，使用下述命令，进入Spark Shell接口，如图35-1所示。

[root@slave1 ~]# /usr/cstor/spark/bin/spark-shell --master spark://master:7077





图35-4 启动master节点

进入Spark Shell命令行执行环境后，依次输入下述代码，即完成模型训练：

import org.apache.spark.mllib.classification.{SVMModel, SVMWithSGD}

import org.apache.spark.mllib.evaluation.BinaryClassificationMetrics

import org.apache.spark.mllib.util.MLUtils

// Load training data in LIBSVM format.

val data = MLUtils.loadLibSVMFile(sc, "/35/in/sample\_libsvm\_data.txt")

// Split data into training (60%) and test (40%).

val splits = data.randomSplit(Array(0.6, 0.4), seed = 11L)

val training = splits(0).cache()

val test = splits(1)

// Run training algorithm to build the model

val numIterations = 100

val model = SVMWithSGD.train(training, numIterations)

// Clear the default threshold.

model.clearThreshold()

// Compute raw scores on the test set.

val scoreAndLabels = test.map { point =>

val score = model.predict(point.features)

(score, point.label)

}

// Get evaluation metrics.

val metrics = new BinaryClassificationMetrics(scoreAndLabels)

val auROC = metrics.areaUnderROC()

println("Area under ROC = " + auROC)

// Save and load model

model.save(sc, "/35/in/scalaSVMWithSGDModel")

val sameModel = SVMModel.load(sc, "/35/in/scalaSVMWithSGDModel")

35.5 实验结果

实验直接使用Spark自带的SVM数据集与SVM代码库，省去大量代码篇幅，读者可下载源码包，然后逐个跟进便可看到完整SVM处理源码。

实验结果是调用训练好的SVM模型对新样本进行分类，也即35.3.2一节，其执行过程如图35-2所示。

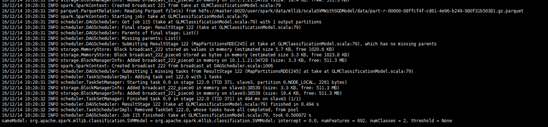


图35-5 实验结果

实验三十六 使用Spark实现FP-Growth

36.1 实验目的

1． 熟练使用Spark Shell接口；

2． 了解FP-Growth关联分析原理；

3． 了解FP-Growth算法执行过程；

4． 配置Spark处理HDFS数据；

5． 使用Spark机器学习包中FP-Growth算法处理HDFS中数据。

36.2 实验要求

使用Spark MLlib（机器学习库）中的FP-Growth算法，处理存储在HDFS上的数据集sample\_fpgrowth.txt。

36.3 实验原理

36.3.1 FP-Growth算法简介

FP的全称是Frequent Pattern，在算法中使用了一种称为频繁模式树（Frequent Pattern Tree）的数据结构。FP-tree是一种特殊的前缀树，由频繁项头表和项前缀树构成。所谓前缀树，是一种存储候选项集的数据结构，树的分支用项名标识，树的节点存储后缀项，路径表示项集。

FP-tree的生成方法如图36-1所示：

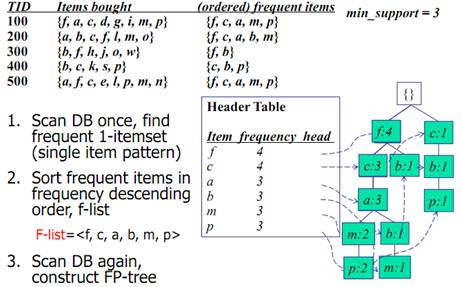


图36-1 FP-tree的生成方法

第二步根据支持度对频繁项进行排序是本算法的关键。第一点是，通过将支持度高的项排在前面，使得生成的FP-tree中，出现频繁的项更可能被共享，从而有效地节省算法运行所需要的空间。另一点是，通过这种排序，可以对FP-tree所包含的频繁模式进行互斥的空间拆分，得到相互独立的子集，而这些子集又组成了完整的信息。

FP-tree子集分割方法

如上图，求p为前缀的投影数据库：根据头表的指针找到FP-tree的两个p节点，搜索出从这两个节点到树的根节点路径节点信息（包含支持度）。然后累加路径节点信息的支持度，删除非频繁项。对剩下的频繁项按照上一节的方法构建FP-tree。过程如图36-2和图36-3所示：

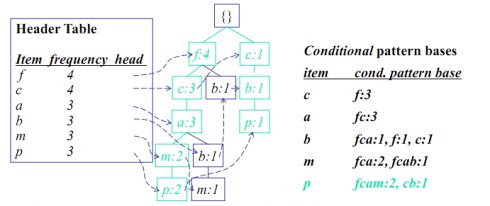


图36-2 搜索节点信息

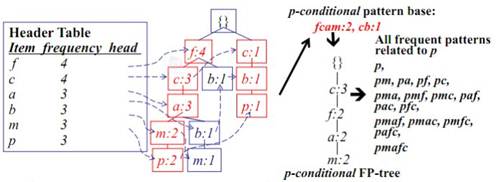


图36-3 构建FP-tree

36.3.2 FP-Growth算法流程

基本思路是：不断地迭代FP-tree的构造和投影过程。

对于每个频繁项，构造它的条件投影数据库和投影FP-tree。对每个新构建的FP-tree重复这个过程，直到构造的新FP-tree为空，或者只包含一条路径。当构造的FP-tree为空时，其前缀即为频繁模式；当只包含一条路径时，通过枚举所有可能组合并与此树的前缀连接即可得到频繁模式。

36.4 实验步骤

关联分析指的是从大规模数据集中寻找物品间的隐含关系，这些关系主要包括“频繁项集”和“关联规则”，挖掘出“频繁项集”后，可从中找出“关联规则”。当使用FP-Growth算法寻找频繁项时，其首先需要构建FP树，接着可通过查找元素项的条件基、构建条件FP树来发现频繁项集，最后再从该“频繁项集”中计算出“关联规则”下述步骤即讲述这一过程。

36.4.1 添加临时JAVA\_HOME环境变量

首先，手工或者使用“一键搭建spark”功能构筑好spark环境。

登录slave1机，添加临时JAVA\_HOME环境变量

[root@slave1 ~]# export JAVA\_HOME=/usr/local/jdk1.7.0\_79

36.4.2 上传训练数据集

查看HDFS里是否已存在目录“/36/in”。若不存在，使用下述命令新建该目录。

[root@slave1 ~]# /usr/cstor/hadoop/bin/hdfs dfs -mkdir -p /36/in

接着，使用下述命令将slave1机本地文件“/root/data/36/sample\_fpgrowth.txt”上传至HDFS的“/36/in”目录：

[root@slave1 ~]# /usr/cstor/hadoop/bin/hdfs dfs -put /root/data/36/sample\_fpgrowth.txt /36/in

最后，请确认HDFS上已经存在文件sample\_fpgrowth.txt。

https://bd.cstor.cn/experiment/exp36.files/image004.jpg

36.4.3 训练SVM模型

准备好输入文件后，下一步便是在Spark集群上执行SVM程序（处理该数据集）。下面的处理代码参考自“http://spark.apache.org/docs/latest/mllib-linear-methods.html”，操作命令主要在slave1机上完成。

首先，在slave1机上，使用下述命令，进入Spark Shell接口，如图36-4所示。

[root@slave1 ~]# /usr/cstor/spark/bin/spark-shell --master spark://master:7077





图36-4 启动master节点

进入Spark Shell命令行执行环境后，依次输入下述代码，即完成模型训练。

import org.apache.spark.mllib.fpm.FPGrowth

import org.apache.spark.rdd.RDD

val data = sc.textFile("data/mllib/sample\_fpgrowth.txt")

val transactions: RDD[Array[String]] = data.map(s => s.trim.split(' '))

val fpg = new FPGrowth()

.setMinSupport(0.2)

.setNumPartitions(10)

val model = fpg.run(transactions)

model.freqItemsets.collect().foreach { itemset =>

println(itemset.items.mkString("[", ",", "]") + ", " + itemset.freq)

}

val minConfidence = 0.8

model.generateAssociationRules(minConfidence).collect().foreach { rule =>

println(

rule.antecedent.mkString("[", ",", "]")

+ " => " + rule.consequent .mkString("[", ",", "]")

+ ", " + rule.confidence)

}

待模型训练结束后，即可在测试数据集上，使用该模型，对测试样本进行分类。

36.5 实验结果

FP-Growth常用于“查看哪些商品经常被一起购买”、“挖掘出哪些新闻广泛被用户浏览过”等场景，当需要使用FP-Growth算法找出数据项中最频繁项集时，首先便是构造FP-Tree，其结果是挖掘出“频繁项集”，如下为实验运行截图，如图36-5和图36-6所示。

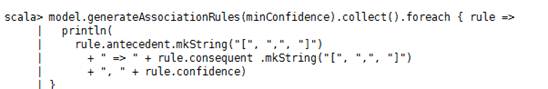


图36-5 实验结果

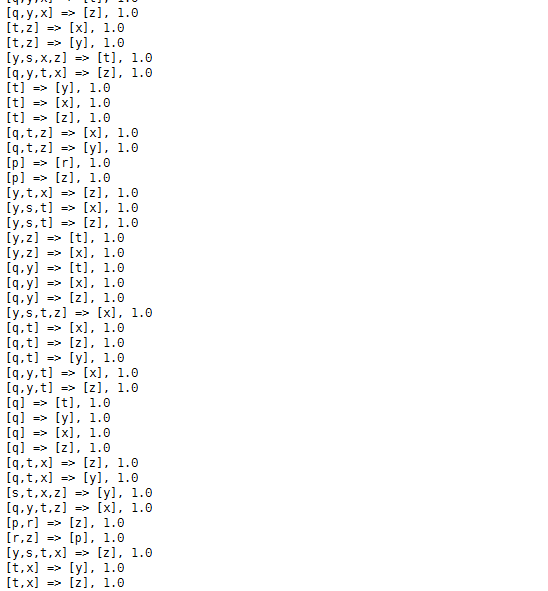


图36-6 实验结果

实验三十七 综合实战：车牌识别

37.1 实验目的

基于MapReduce思想，编写车牌识别程序。实现对江苏某两处监控图片中的车牌的识别，完成对除江苏省车辆外的外省车辆的统计，并完成对两处监控中套牌车辆的识别，并得出结果。

37.2 实验要求

要能理解MapReduce编程思想，会编写MapReduce车牌识别程序，能够利用Java调用已封装好C++的so动态库完成车牌的识别，然后利用MapReduce完成对车牌的统计和对套牌车的识别。最后将其执行并分析执行过程。

37.3 实验步骤

37.3.1 编写程序

对于外省车牌的统计，我们可以理解为WordCount程序，利用WordCount的思想，在Reduce阶段对“苏”车牌不进行输出即可。

对于套牌车辆的识别，Map阶段与WordCount是一样的，在Reduce阶段我们将统计相同车牌出现的情况。

我们将编写两个类，一个类用于加载C++的so动态库，另一个完成MapReduce算法。

SelectPic.java

import java.io.File;

import java.util.Vector;

public class SelectPic {

private Vector<String> pics = null;

public SelectPic() {

pics = new Vector<>();

}

static {

System.loadLibrary("Easyper");

}

public native String getPlate(String inPath);

public void doPicNames(String inPath) {

File file = new File(inPath);

File[] listFiles = file.listFiles();

for (File files : listFiles) {

if (!files.isDirectory()) {

String name = files.getName();

String plate = getPlate(inPath + "/" + name);

if (plate.length() > 3)

plate.substring(3);

}

}

}

public static void main(String[] args) {

SelectPic sp = new SelectPic();

sp.doPicNames("image");

}

}

PlateRecog.java

import java.io.IOException;

import java.util.Vector;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.FileSystem;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.NullWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.io.WritableComparable;

import org.apache.hadoop.io.WritableComparator;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.Reducer;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import org.apache.hadoop.util.GenericOptionsParser;

public class PlateRecog {

public static class PlateRecogMapper\_coll extends Mapper<LongWritable, Text, Text, IntWritable> {

private SelectPic sp = null;

private FileSystem HDFS = null;

@Override

protected void setup(Mapper<LongWritable, Text, Text, IntWritable>.Context context)

throws IOException, InterruptedException {

Configuration conf = context.getConfiguration();

sp = new SelectPic();

HDFS = FileSystem.get(conf);

}

@Override

protected void map(LongWritable key, Text value, Mapper<LongWritable, Text, Text, IntWritable>.Context context)

throws IOException, InterruptedException {

String[] splits = value.toString().split(" ");

String ds = splits[0].substring(0, splits[0].length() - 4) + "\_t.jpg";

Path src = new Path(splits[0]);

Path dst = new Path(ds);

HDFS.copyToLocalFile(src, dst);

String plate = sp.getPlate(ds);

plate = plate.substring(3);

context.write(new Text(plate.substring(0, 1)), new IntWritable(1));

}

}

// 统计非江苏的车牌的数量

public static class PlateRecogReducer\_coll extends Reducer<Text, IntWritable, Text, IntWritable> {

@Override

protected void reduce(Text k2, Iterable<IntWritable> v2s,

Reducer<Text, IntWritable, Text, IntWritable>.Context context)

throws IOException, InterruptedException {

if (!k2.toString().equals("苏")) {

int sum = 0;

for (IntWritable v2 : v2s) {

sum += v2.get();

}

context.write(new Text(k2.toString() + " 车牌的数量："), new IntWritable(sum));

}

}

}

public static class PlateRecogMapper\_find extends Mapper<LongWritable, Text, Text, Text> {

private SelectPic sp = null;

private FileSystem HDFS = null;

@Override

protected void setup(Mapper<LongWritable, Text, Text, Text>.Context context)

throws IOException, InterruptedException {

Configuration conf = context.getConfiguration();

sp = new SelectPic();

HDFS = FileSystem.get(conf);

}

@Override

protected void map(LongWritable key, Text value, Mapper<LongWritable, Text, Text, Text>.Context context)

throws IOException, InterruptedException {

String[] splits = value.toString().split(" ");

String ds = splits[0].substring(0, splits[0].length() - 4) + "\_t.jpg";

Path src = new Path(splits[0]);

Path dst = new Path(ds);

HDFS.copyToLocalFile(src, dst);

String plate = sp.getPlate(ds);

plate = plate.substring(3);

String plateInfo = plate + '\_' + splits[1];

// 皖A12345\_1 45

context.write(new Text(plateInfo), new Text(dst.getName().substring(0, dst.getName().length() - 6)));

}

}

public static class PlateRecogReducer\_find extends Reducer<Text, Text, Text, NullWritable> {

private Vector<String> samePlates = null;

@Override

protected void setup(Reducer<Text, Text, Text, NullWritable>.Context context)

throws IOException, InterruptedException {

samePlates = new Vector<>();

}

@Override

protected void reduce(Text k2, Iterable<Text> v2s, Reducer<Text, Text, Text, NullWritable>.Context context)

throws IOException, InterruptedException {

int sum = 0;

String[] splits = k2.toString().split("\_");

samePlates.clear();

for (Text v2 : v2s) {

++sum;

samePlates.add(v2.toString());

}

if (sum > 1) {

String sameplate = "";

for (int i = 0; i < samePlates.size() - 1; ++i) {

sameplate += samePlates.get(i) + " 和 ";

}

sameplate += samePlates.get(samePlates.size() - 1);

context.write(new Text("存在套牌的车辆的车牌号为 :" + splits[0] + " 车牌编号为： " + sameplate), NullWritable.get());

}

}

}

public static class findGroupComparator extends WritableComparator {

protected findGroupComparator() {

super(Text.class, true);// 注册comparator

}

@Override

public int compare(WritableComparable a, WritableComparable b) {

// TODO Auto-generated method stub

Text ti1 = (Text) a;

Text ti2 = (Text) b;

return ti1.toString().substring(0, 7).compareTo(ti2.toString().substring(0, 7));

}

}

public static void main(String[] args) {

Configuration conf = new Configuration();

try {

GenericOptionsParser goparser = new GenericOptionsParser(conf, args);

String otherargs[] = goparser.getRemainingArgs();

Job job\_coll = Job.getInstance(conf);

job\_coll.setJarByClass(PlateRecog.class);

job\_coll.setMapperClass(PlateRecogMapper\_coll.class);

job\_coll.setReducerClass(PlateRecogReducer\_coll.class);

job\_coll.setMapOutputKeyClass(Text.class);

job\_coll.setMapOutputValueClass(IntWritable.class);

FileInputFormat.addInputPath(job\_coll, new Path(otherargs[0]));

FileOutputFormat.setOutputPath(job\_coll, new Path(otherargs[1]));

if ((job\_coll.waitForCompletion(true) ? 1 : 0) == 1) {

Job job\_find = Job.getInstance(conf);

job\_find.setJarByClass(PlateRecog.class);

job\_find.setMapperClass(PlateRecogMapper\_find.class);

job\_find.setReducerClass(PlateRecogReducer\_find.class);

job\_find.setMapOutputKeyClass(Text.class);

job\_find.setMapOutputValueClass(Text.class);

job\_find.setOutputKeyClass(Text.class);

job\_find.setOutputValueClass(NullWritable.class);

// job\_find.setPartitionerClass(findParitioner.class);

job\_find.setGroupingComparatorClass(findGroupComparator.class);

FileInputFormat.addInputPath(job\_find, new Path(otherargs[0]));

FileOutputFormat.setOutputPath(job\_find, new Path(otherargs[2]));

job\_find.waitForCompletion(true);

}

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (ClassNotFoundException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

37.3.2 环境准备

将实验数据上传到HDFS上，包括两部分，一部分是车牌照片和车牌对应的plate.txt文件，另一部分是所有车牌的照片。

在client机器上在Hadoop的HDFS中创建两个目录。

[root@client hadoop]# bin/hadoop fs -mkdir -p /user/mapreduce/platerecog/in

[root@client hadoop]# bin/hadoop fs -mkdir /user/mapreduce/platerecog/images

上传数据。

[root@client hadoop]# bin/hadoop fs -put /root/data/37/images/\* /user/mapreduce/platerecog/images

[root@client hadoop]# bin/hadoop fs -put /root/data/37/in/plate.txt /user/mapreduce/platerecog/in

在master机器上，将动态库所需要的配置文件拷贝至Hadoop根目录下。

[root@master hadoop]# \cp -r ../plate/use/etc/ ./

[root@master hadoop]# \cp -r ../plate/use/resources/ ./

[root@master hadoop]# cp -r ../plate/use/libEasyper.so ./lib

然后将动态库和配置文件拷贝至slave1，slave2，slave3上。

先创建 machines文件写入几台机器的主机名。

[root@ master hadoop]# vim machines

slave1

slave2

slave3

client

然后利用for进行远程拷贝。

[root@master hadoop]# for x in `cat machines`;do scp -r ../plate/use/etc/ $x:/usr/cstor/hadoop/;done;

[root@master hadoop]# for x in `cat machines`;do scp -r ./resources/ $x:/usr/cstor/hadoop/;done;

[root@master hadoop]# for x in `cat machines`;do scp -r ../plate/use/libEasyper.so \

> $x:/usr/cstor/hadoop/lib;done;

37.3.3 打包提交

然后使用Eclipse开发工具将该代码打包，选择主类为PlateRecog。假定打包后的文件名为PlateRecog.jar，主类PlateRecog位于默认包下，则可使用如下命令向Hadoop集群提交本应用。

[root@client hadoop]# bin/hadoop jar PlateRecog.jar -files \

> ../plate/use/libEasyper.so,../plate/use/etc/,../plate/use/resources/ \

> /user/mapreduce/platerecog/in/plate.txt /user/mapreduce/platerecog/count/ \

> /user/mapreduce/platerecog/samePlate/

其中“hadoop”为命令，“jar”为命令参数，后面紧跟打的包，“-files”是附加文件指令，“../plate/use/libEasyper.so,../plate/use/etc/,../plate/use/resources/”是同时提交到集群上的配置文件和动态库。“/user/mapreduce/platerecog/in/plate.txt”为输入文件在HDFS中的位置，“/user/mapreduce/platerecog/count/”和 “/user/mapreduce/platerecog/samePlate/”为输出文件在HDFS中的位置，分别对应的是统计车牌的数量的输出文件count，套牌车识别的结果文件samePlate。

37.4 实验结果

37.4.1 输入数据

输入数据格式如下：plate.txt (空格（' '）分割)（数据放在/root/data/37/in目录下）

/user/mapreduce/platerecog/images/99.jpg 2

/user/mapreduce/platerecog/images/100.jpg 2

/user/mapreduce/platerecog/images/101.jpg 2

/user/mapreduce/platerecog/images/102.jpg 2

/user/mapreduce/platerecog/images/103.jpg 2

/user/mapreduce/platerecog/images/104.jpg 2

/user/mapreduce/platerecog/images/105.jpg 2

/user/mapreduce/platerecog/images/106.jpg 2

/user/mapreduce/platerecog/images/107.jpg 2

······

37.4.2 执行结果

在client上执行对HDFS上的文件/user/mapreduce/platerecog/count/part-r-00000内容查看的操作，查看外省车辆的统计情况

[root@client hadoop]# bin/hadoop fs -cat /user/mapreduce/platerecog/count/p\*

屏幕上显示如图37-1所示。



图37-1 查看外省车牌统计情况

在client上执行对HDFS上的文件/user/mapreduce/platerecog/samePlate/part-r-00000内容查看的操作，查看两处监控的套牌车辆的情况。

[root@client hadoop]# bin/hadoop fs -cat /user/mapreduce/platerecog/samePlate/p\*

屏幕上显示如图37-2所示。

https://bd.cstor.cn/experiment/exp37.files/image002.jpg

图37-2 查看套牌情况

我们将两个编号的车牌图片拿出来查看一下，如图37-3所示。



图37-3 查看两个编号的车牌照片

实验三十八 综合实战：搜索引擎

38.1 实验目的

1． 利用大数据实验环境完成一个真实的项目；

2． 结合多个大数据的组件练习如何在实际项目中使用大数据的组件；

3． 能使用分布式的思想对数据进行清洗，处理；

4． 考虑大数据环境对数据的处理方法进行优化；

5． 利用大数据的思路，了解通用的搜索引擎技术流程。

38.2 实验要求

在实验结束的时候能达到实验结果所展示的要求，能从HBase中正确查询到所需要的数据，并且结果能按照相关性大小排序。

能明白HBase建立索引的理由，并能理解HBase建立索引的方法，能正确使用索引，利用索引在HBase中作用。

能理解在扒取互联网数据的时候的流程，在实验完成之后有兴趣可以对其他网站运用同样的方法对数据进行扒取。

能完整的将整个实验流程理解。了解大数据环境下利用HBase组件的一些功能。并能比较利用NoSQL的数据库和关系型数据库的区别。

38.3 实验步骤

我们的实验的内容主要是先从淘宝的网页中扒取部分商品的信息，然后从这些信息中提取去我们所需要的几项数据，将其存储在HBase中。然后我们可以通过关键词搜索从HBase中将这些数据根据我们的关键词提取出来。首先，我们需要在准备好的HBase中创建数据表goodinfo和索引表indexinfo。

建表语句：

hbase(main):001:0> create ‘goodinfo’,’info’

hbase(main):002:0> create ‘indexinfo’,’info’

38.3.1 新建Java项目

因为HBase只能使用rowKey从HBase中读取数据。而我们需要使用关键词从HBase读取数据，所以我们需要对商品的标题中提取关键词，然后对关键词和URL建立索引。这样我们就可以通过商品的的标题分词过后的关键词进行搜索查询到商品的URL，然后通过URL去取得商品的信息。

新建createindex的项目，主要分为几个步骤，新建Hadoop任务的Map类和Reduce类，新建任务类。

项目的目录结构如图38-1所示：

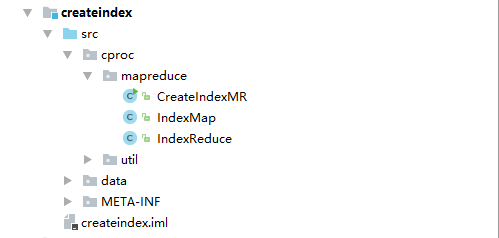


图38-1 createindex项目目录结构

项目所需的jar可在实验的/root/data/38/jar使用ftp下载到本地即可。

新建IndexMap类。Map类的代码为：

package cproc.mapreduce;  
  
import java.io.IOException;  
import java.util.ArrayList;  
import java.util.HashMap;  
import java.util.List;  
import java.util.Map;  
  
import org.apache.hadoop.hbase.client.Result;  
import org.apache.hadoop.hbase.io.ImmutableBytesWritable;  
import org.apache.hadoop.hbase.mapreduce.TableMapper;  
import org.apache.hadoop.hbase.util.Bytes;  
import org.apache.hadoop.io.Text;  
import org.apache.hadoop.mapreduce.Mapper;  
  
import cproc.util.BuildSegmenter;  
import edu.stanford.nlp.ie.crf.CRFClassifier;  
import edu.stanford.nlp.ling.CoreLabel;  
  
public class IndexMap extends TableMapper<Text, Text> {  
   
 CRFClassifier<CoreLabel> segmenter = null;  
   
 @Override  
 protected void setup(  
 Mapper<ImmutableBytesWritable, Result, Text, Text>.Context context)  
 throws IOException, InterruptedException {  
 super.setup(context);  
   
 // 初始化分词器 —— 加载速度慢，该步骤仅操作一次，耗时约10s  
 if(segmenter==null){  
 BuildSegmenter bs = new BuildSegmenter();  
 segmenter = bs.BuildSegmenter();  
 }  
 }  
  
 @Override  
 public void map(ImmutableBytesWritable key, Result value, Context context)  
 throws IOException, InterruptedException {  
   
 String URL = Bytes.toString(key.get());  
 Map<String, Integer> wordmap = null;  
   
 String title = "";  
 byte[] bytes = value.getValue(Bytes.toBytes("info"), Bytes.toBytes("itemTitle"));  
 if(bytes!=null && bytes.length>0){  
 title = Bytes.toString(bytes);   
 }  
   
 //标题分词  
 List<String> keyWordSeg = new ArrayList<String>();  
 keyWordSeg = segmenter.segmentString(title);  
   
 //每个URL的标题内单词合并  
 wordmap = new HashMap<String, Integer>();  
 for(String word : keyWordSeg){  
 if(wordmap.containsKey(word)){  
 wordmap.put(word, (Integer)wordmap.get(word)+1);  
 }else{  
 wordmap.put(word, 1);  
 }  
 }  
   
 for(String word : wordmap.keySet()){  
 context.write(new Text(word),new Text(wordmap.get(word)+"##"+URL));   
 }  
   
 }  
}

新建IndexReduce类，Reduce类的代码如下：

package cproc.mapreduce;  
  
import java.io.IOException;  
  
import org.apache.hadoop.hbase.client.Put;  
import org.apache.hadoop.hbase.io.ImmutableBytesWritable;  
import org.apache.hadoop.hbase.mapreduce.TableReducer;  
import org.apache.hadoop.hbase.util.Bytes;  
import org.apache.hadoop.io.Text;  
  
public class IndexReduce extends TableReducer<Text, Text, ImmutableBytesWritable>  
{  
   
 @Override  
 protected void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException   
 {  
 String str = "";  
   
 for (Text value : values) {  
 str += value.toString() + "####";  
 }  
   
 if(str!=null && str.length()>4)  
 {  
 str = str.substring(0, str.length()-4);  
 }  
 Put put = null;  
 try{  
 put = new Put(Bytes.toBytes(key.toString()));  
  
 }catch (Exception e){  
 System.out.println("key is : "+ key.toString());  
 System.out.println(e.getMessage());  
 }  
 if(put!=null) {  
 // 列簇 , 列名, 值  
 put.addColumn(Bytes.toBytes("info"), Bytes.toBytes("val"), Bytes.toBytes(str));  
  
 context.write(new ImmutableBytesWritable(Bytes.toBytes(key.toString())), put);  
 }  
  
 }  
   
}

新建CreateIndexMR，新建MapReduce的主方法。

package cproc.mapreduce;  
  
import org.apache.hadoop.conf.Configuration;  
import org.apache.hadoop.filecache.DistributedCache;  
import org.apache.hadoop.fs.Path;  
import org.apache.hadoop.hbase.HBaseConfiguration;  
import org.apache.hadoop.hbase.client.Scan;  
import org.apache.hadoop.hbase.mapreduce.TableMapReduceUtil;  
import org.apache.hadoop.hdfs.DistributedFileSystem;  
import org.apache.hadoop.io.Text;  
import org.apache.hadoop.mapreduce.Job;  
  
  
public class CreateIndexMR {  
  
 @SuppressWarnings("deprecation")  
 public static void main(String[] args) throws Exception {  
  
 //初始化conf文件  
 Configuration conf = HBaseConfiguration.create();  
 //HBase配置  
 conf.set("hbase.zookeeper.quorum", "slave1:2181,slave2:2181,slave3:2181");//你的Zookeeper的地址  
 conf.set("zookeeper.znode.parent", "/hbase");  
   
 //定义任务  
 Job job = new Job(conf, "createIndex");  
 job.setJarByClass(CreateIndexMR.class);  
   
   
 Scan scan = new Scan();   
 scan.setCaching(500); // 默认是1，应该设置的大一些  
 scan.setCacheBlocks(false); // 当mr或者全表查询的时候要设置成false，查询热数据的的时候可以使用，这样就在HBase的jvm中缓存当前记录  
   
 //设置map  
 TableMapReduceUtil.initTableMapperJob(   
 "goodinfo", // input table  
 scan, // Scan   
 IndexMap.class, // mapper class  
 Text.class, // mapper output key  
 Text.class, // mapper output value   
 job);   
   
 //设置reduce (output table , reducer class , job)  
 TableMapReduceUtil.initTableReducerJob("indexinfo", IndexReduce.class, job);  
   
 // 运行job  
 System.exit(job.waitForCompletion(true) ? 0 : 1);  
   
 }  
  
}

新建BuildSegmenter类，新建分词工具类。

package cproc.util;  
  
import java.util.Properties;  
  
import edu.stanford.nlp.ie.crf.CRFClassifier;  
import edu.stanford.nlp.ling.CoreLabel;  
  
/\*\*  
 \* 初始化分词器  
 \*/  
  
public class BuildSegmenter {  
 public CRFClassifier<CoreLabel> BuildSegmenter() {  
 /\*\*  
 \* 初始化分词器  
 \* @return  
 \*/  
 //String basedir = System.getProperty("Sousuo", "data");  
// String basedir = BuildSegmenter.class.getClassLoader().getResource("data").getPath();  
 String basedir = "/data";  
   
 Properties props = new Properties();  
 props.setProperty("sighanCorporaDict", basedir);  
 props.setProperty("serDictionary", basedir + "/dict-chris6.ser.gz");  
   
 props.setProperty("inputEncoding", "UTF-8");  
 props.setProperty("sighanPostProcessing", "true");  
   
 CRFClassifier<CoreLabel> segmenter = new CRFClassifier<>(props);  
 System.out.println(basedir);  
 segmenter.loadClassifierNoExceptions(basedir + "/ctb.gz", props);  
   
 return segmenter;  
 }  
}

到此，CreateIndex代码部分已经完成。Createindex项目的作用是对HBase 中的goodinfo集合数据建立索引表indexinfo。

38.3.2 新建JavaWeb项目

该项目的主要作用是提供Web界面扒取淘宝的数据和搜索数据。该项目部署在本地服务器上。

该项目的目录结构如图38-2所示：

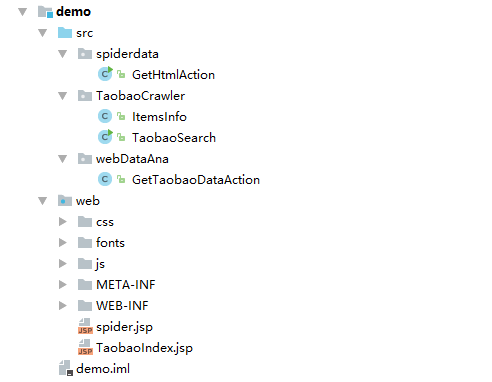
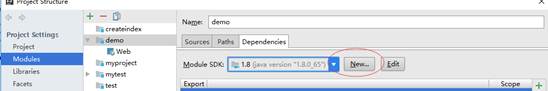


图38-2 JavaWeb项目目录结构

对于Web目录下的除jsp页面外的代码放在/root/data/38/web 目录下，请读者自行下载加入自己的目录下。

该项目中存在分词器的工具，该工具是必须使用jdk1.8版本才能正确运行，所以在创建项目之前需要安装jdk1.8到本地，该 javaWeb项目需要依赖于该jdk。请自行修改，修改步骤如下。

点击File->project Structure -> 选择modules -> 点击新建的javaWeb项目-> 点击如下位置

选择安装的jdk1.8 的目录即可。

新建spider.jsp文件，作为页面扒取的Web页面。

<!-- you\_shoubian -->  
<%@page import="TaobaoCrawler.ItemsInfo"%>  
<%@page import="Java.util.\*"%>  
<%@ page import="WebDataAna.GetTaobaoDataAction" %>  
<%@ page language="Java" contentType="text/html; charset=UTF-8"  
 pageEncoding="UTF-8"%>  
  
  
<%  
 String searchWord = "";  
 String pageNumStr = "";  
 String itemLen = "";  
  
 String msg = "";  
%>  
  
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">  
<html>  
<head>  
 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">  
 <title>淘宝集合</title>  
 <link rel="stylesheet" href="css/bootstrap.min.css">  
 <link rel="stylesheet" href="css/myTaobaoCss.css">  
 <script type="text/Javascript" src="js/jquery-1.11.3.min.js"></script>  
 <script type="text/Javascript" src="js/bootstrap.min.js"></script>  
</head>  
<body>  
<div class="container">  
 <div style="height: 100px"></div>  
  
 <div class="formStyle col-md-offset-1 col-md-10 ">  
 <form class="" method="post" action="/getHtml">  
 <div class="form-group col-md-7 mySearchMar">  
 <input type="text" class="form-control " id="searchWords"  
 name="searchWords" placeholder="请输入搜索的内容,多个关键词用 , 隔开">  
 </div>  
 <div class="form-group col-md-2 myPageMar">  
 <div class="input-group">  
 <input type="text" class="form-control" id="pageNum" name="pageNum"  
 placeholder="1">  
 <div class="input-group-addon">页</div>  
 </div>  
 </div>  
  
  
 <button type="submit" class="btn btn-default col-md-1"  
 onclick="showMsg()">检索</button>  
 </form>  
 </div>  
  
 <div class="tableStyle col-md-offset-1 col-md-10">  
 <div id="msg"></div>  
 </div>  
  
  
 <%  
 if (request.getAttribute("searchWord") != null) {  
 searchWord = (String) request.getAttribute("searchWord");  
 itemLen = (String) request.getAttribute("itemLen");  
 String pageNum = (String) request.getAttribute("pageNumStr");  
  
 msg = "共获取 <b>" + pageNum + "</b> 页,获取数据 <b>" + itemLen + "</b> 条,搜索词为 <b>"+searchWord+"</b>";  
 if(searchWord.equals("无搜索词，结果随机搜索")){  
 searchWord="";  
 }  
  
 }  
 %>  
  
  
</div>  
  
<script type="text/Javascript">  
 $("#searchWord").val("<%=searchWord%>");  
 $("#pageNum").val("<%=pageNumStr%>");  
 $("#msg").html("<%=msg%>");  
  
 function showMsg() {  
 $("#msg").text("正在检索...");  
 }  
</script>  
</body>  
</html>

新建TaobaoIndex.jsp 文件，作为Taobao数据展示Web页面。

<!-- you\_shoubian -->  
<%@page import="TaobaoCrawler.ItemsInfo"%>  
<%@page import="java.util.\*"%>  
<%@ page import="WebDataAna.GetTaobaoDataAction" %>  
<%@ page language="Java" contentType="text/html; charset=UTF-8"  
 pageEncoding="UTF-8"%>  
  
  
<%  
 String searchWord = "";  
 String pageNumStr = "";  
 String msg = "";  
%>  
  
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">  
<html>  
<head>  
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">  
<title>淘宝集合</title>  
<link rel="stylesheet" href="css/bootstrap.min.css">  
<link rel="stylesheet" href="css/myTaobaoCss.css">  
<script type="text/Javascript" src="js/jquery-1.11.3.min.js"></script>  
<script type="text/Javascript" src="js/bootstrap.min.js"></script>  
</head>  
<body>  
 <div class="container">  
 <div style="height: 100px"></div>  
  
 <div class="formStyle col-md-offset-1 col-md-10 ">  
 <form class="" method="post" action="/getTaobaoData">  
 <div class="form-group col-md-7 mySearchMar">  
 <input type="text" class="form-control " id="searchWord"  
 name="searchWord" placeholder="请输入搜索的内容">  
 </div>  
 <div class="form-group col-md-2 myPageMar">  
 <div class="input-group">  
 <input type="text" class="form-control" id="pageNum" name="pageNum"  
 placeholder="1">  
 <div class="input-group-addon">项</div>  
 </div>  
 </div>  
  
  
 <button type="submit" class="btn btn-default col-md-1"  
 onclick="showMsg()">检索</button>  
 </form>  
 </div>  
  
 <div class="tableStyle col-md-offset-1 col-md-10">  
 <div id="msg"></div>  
 </div>  
  
 <div class="tableStyle col-md-offset-1 col-md-10">  
 <table class="table table-striped">  
 <thead>  
 <tr>  
 <th>#</th>  
 <th>商品名</th>  
 <th class="col-md-1">价格</th>  
 <th>交易量</th>  
 <th class="col-md-2">店铺</th>  
 <th>所在地</th>  
 </tr>  
 </thead>  
 <tbody>  
 <%  
 List<ItemsInfo> itemsInfoList = new ArrayList<ItemsInfo>();  
 if (request.getAttribute("itemsInfoList") != null) {  
 itemsInfoList = (ArrayList<ItemsInfo>) request.getAttribute("itemsInfoList");  
 searchWord = (String) request.getAttribute("searchWord");  
 pageNumStr = (String) request.getAttribute("pageNumStr");  
 int pageNum = (Integer) request.getAttribute("pageNum");  
   
 msg = "共获取数据 <b>" + itemsInfoList.size() + "</b> 条,搜索词为 <b>"+searchWord+"</b>";  
 if("无搜索词，结果随机搜索".equals(searchWord)){  
 searchWord="";  
 }  
  
 int len = itemsInfoList.size();  
 for (int i = 0; i < len; i++) {  
 %>  
 <tr>  
 <td><%=i + 1%></td>  
 <td><a href="<%=itemsInfoList.get(i).getItemAddress()%>"  
 title="点击查看详情"><%=itemsInfoList.get(i).getItemTitle()%></a></td>  
 <td><%=itemsInfoList.get(i).getItemPrice()%></td>  
 <td><%=itemsInfoList.get(i).getItemDealCnt()%></td>  
 <td><%=itemsInfoList.get(i).getItemShop()%></td>  
 <td><%=itemsInfoList.get(i).getItemShopLocaltion()%></td>  
 </tr>  
 <%  
 }  
 }  
 %>  
   
 </table>  
 </div>  
  
 </div>  
  
 <script type="text/Javascript">  
 $("#searchWord").val("<%=searchWord%>");  
 $("#pageNum").val("<%=pageNumStr%>");  
 $("#msg").html("<%=msg%>");  
  
 function showMsg() {  
 $("#msg").text("正在检索...");  
 }  
 </script>  
</body>  
</html>

创建servlet类，GetHtmlAction类，这个servlet访问/getHtml的http请求。

package spiderdata;

import java.io.File;

import java.io.IOException;

import java.util.ArrayList;

import java.util.List;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.hbase.HBaseConfiguration;

import org.apache.hadoop.hbase.TableName;

import org.apache.hadoop.hbase.client.Connection;

import org.apache.hadoop.hbase.client.ConnectionFactory;

import org.apache.hadoop.hbase.client.Put;

import org.apache.hadoop.hbase.client.Table;

import org.apache.hadoop.hbase.util.Bytes;

import org.jsoup.Jsoup;

import org.jsoup.nodes.Document;

import org.jsoup.nodes.Element;

import org.jsoup.select.Elements;

import TaobaoCrawler.ItemsInfo;

import edu.stanford.nlp.ie.crf.CRFClassifier;

import edu.stanford.nlp.ling.CoreLabel;

/\*\*

\* Servlet implementation class GetTaobaoDataAction

\*/

@WebServlet("/getHtml")

public class GetHtmlAction extends HttpServlet {

private static final long serialVersionUID = 1L;

static String jsFile = "D:\\casperJS\\UrlCatch.js";

static String tempfile = "D:\\tempHTML\\temp.html";

static List<ItemsInfo> itemsInfoList = new ArrayList<ItemsInfo>();

CRFClassifier<CoreLabel> segmenter = null;

//HBase

private static final Configuration configuration = HBaseConfiguration.create();

private static Connection connection;

static {

configuration.set("hbase.zookeeper.quorum", " slave1:2181,slave2:2181,slave3:2181");

configuration.set("zookeeper.znode.parent", "/hbase");

try {

connection = ConnectionFactory.createConnection(configuration);

} catch (IOException e) {

e.printStackTrace();

}

}

public GetHtmlAction() {

super();

}

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

doPost(request, response);

}

protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

// TODO Auto-generated method stub

response.setContentType("text/html;charset=UTF-8");

response.setHeader("content-type", "text/html;charset=utf-8");

response.setCharacterEncoding("UTF-8");

try {

String searchWords = new String(

request.getParameter("searchWords")

.getBytes("iso-8859-1"), "utf-8");

String[] listWords = searchWords.split(",");

String pageNumStr = new String(request.getParameter("pageNum").getBytes("iso-8859-1"), "utf-8");

int pageNum = 1;

try {

pageNum = Integer.parseInt(pageNumStr);

} catch (NumberFormatException e) {

System.out.println("页数不是整数，默认为1");

pageNum = 1;

}

if(listWords==null || listWords.length <= 0){

System.out.println("无搜索词");

return;

}

int all = saveGoods(listWords,pageNum);

request.removeAttribute("searchWord");

request.removeAttribute("itemLen");

request.removeAttribute("pageNumStr");

request.setAttribute("searchWord",searchWords.toString());

request.setAttribute("itemLen",all+"");

request.setAttribute("pageNumStr",pageNum+"");

request.getRequestDispatcher("spider.jsp").forward(request, response);

} catch (Exception e) {

e.printStackTrace();

}

}

public int saveGoods(String[] ListWords, int pageNum) throws Exception

{

int sum = 0;

for(String searchWord : ListWords)

{

System.out.println(searchWord);

boolean flag = false;

//初始化每页商品个数

int cnt = 1;

//初始化从第s个商品开始爬取

int s = 1;

for (int i = 1; i <= pageNum; i++) {

System.out.println("正在获取第" + i + "/" + pageNum + "页...");

Document doc = getHTMLDoc(jsFile, searchWord, s);

itemsInfoList.clear();

getItemsInfo(doc);

if (!flag) {

cnt = itemsInfoList.size();

flag = true;

}

s = i \* cnt;

//保存HBase

TableName tbname = TableName.valueOf("goodinfo");

Table table = connection.getTable(tbname);

//商品对象集合

List<Put> putList = new ArrayList<Put>();

Put put = null;

for(ItemsInfo info : itemsInfoList)

{

sum ++;

//商品信息

put = new Put(info.getItemAddress().getBytes());

put.addColumn(Bytes.toBytes("info"),

Bytes.toBytes("itemTitle"),

Bytes.toBytes(info.getItemTitle()));

put.addColumn(Bytes.toBytes("info"),

Bytes.toBytes("itemShop"),

Bytes.toBytes(info.getItemShop()));

put.addColumn(Bytes.toBytes("info"),

Bytes.toBytes("itemShopLocaltion"),

Bytes.toBytes(info.getItemShopLocaltion()));

put.addColumn(Bytes.toBytes("info"),

Bytes.toBytes("itemPrice"),

Bytes.toBytes(info.getItemPrice()));

put.addColumn(Bytes.toBytes("info"),

Bytes.toBytes("itemDealCnt"),

Bytes.toBytes(info.getItemDealCnt()));

putList.add(put);

}

table.put(putList);

table.close();

}

}

return sum;

}

/\* 得到html页面 \*/

public static Document getHTMLDoc(String jsFile, String searchWord, int s) {

String tempfile = "D:\\tempHTML\\temp.html";

try {

String command = "cmd /c casperjs "+jsFile + " --url=https://s.taobao.com/search?q="

+ searchWord + "^&s=" + s + " --tempfile=d:\\tempHTML\\temp.html";

System.out.println(command);

Process process = Runtime.getRuntime().exec(command);

process.waitFor();

// 获取html...

File htmlfile = new File(tempfile);

Document doc = Jsoup.parse(htmlfile, "UTF-8");

return doc;

} catch (Exception e) {

e.printStackTrace();

return null;

}

}

/\* 获取宝贝信息 \*/

public static void getItemsInfo(Document doc) {

Elements elements = doc.getElementsByClass("ctx-box");

for (Element el : elements) {

ItemsInfo item = new ItemsInfo();

String ItemTitle = el.getElementsByClass("title").first().text();

String ItemShop = el.getElementsByClass("shop").first().text();

String ItemShopLocaltion = el.getElementsByClass("location").first().text();

String ItemPrice = el.getElementsByClass("price").first().text();

String ItemDealCnt = el.getElementsByClass("deal-cnt").first().text();

Element alinkElement = el.getElementsByClass("title").first();

String ItemAddress = alinkElement.getElementsByTag("a").attr("href");

if (ItemAddress.startsWith("//")) {

ItemAddress = "https:" + ItemAddress;

}

item.setItemTitle(ItemTitle);

item.setItemShop(ItemShop);

item.setItemShopLocaltion(ItemShopLocaltion);

item.setItemPrice(ItemPrice);

item.setItemDealCnt(ItemDealCnt);

item.setItemAddress(ItemAddress);

itemsInfoList.add(item);

}

}

public static void main(String[] args) throws Exception {

String ss = "手机壳,硬盘";

String[] ListWords = ss.split(",");

int pageNum = 2;

GetHtmlAction g = new GetHtmlAction();

g.saveGoods(ListWords, pageNum);

}

}

创建ItemInfo类，作为从数据库读取数据的model类。

package TaobaoCrawler;  
/\*\*  
 \* author:you\_shoubian time:2016-1-19  
 \*/  
public class ItemsInfo {  
   
 String itemTitle; //商品标题(名称)  
 String itemShop; //商品店铺名  
 String itemShopLocaltion; //商品店铺所在地  
 String itemPrice; //商品价格  
 String itemDealCnt; //商品交易量  
 String itemAddress; //商品详细地址  
   
   
   
 public String getItemTitle() {  
 return itemTitle;  
 }  
 public void setItemTitle(String itemTitle) {  
 this.itemTitle = itemTitle;  
 }  
 public String getItemShop() {  
 return itemShop;  
 }  
 public void setItemShop(String itemShop) {  
 this.itemShop = itemShop;  
 }  
 public String getItemShopLocaltion() {  
 return itemShopLocaltion;  
 }  
 public void setItemShopLocaltion(String itemShopLocaltion) {  
 this.itemShopLocaltion = itemShopLocaltion;  
 }  
 public String getItemPrice() {  
 return itemPrice;  
 }  
 public void setItemPrice(String itemPrice) {  
 this.itemPrice = itemPrice;  
 }  
 public String getItemDealCnt() {  
 return itemDealCnt;  
 }  
 public void setItemDealCnt(String itemDealCnt) {  
 this.itemDealCnt = itemDealCnt;  
 }  
 public String getItemAddress() {  
 return itemAddress;  
 }  
 public void setItemAddress(String itemAddress) {  
 this.itemAddress = itemAddress;  
 }  
   
   
}

创建Taobaosearch类，用于从HBase中读取淘宝数据。

package TaobaoCrawler;  
  
import org.apache.hadoop.conf.Configuration;  
import org.apache.hadoop.HBase.HBaseConfiguration;  
import org.apache.hadoop.HBase.TableName;  
import org.apache.hadoop.HBase.client.\*;  
import org.apache.hadoop.HBase.util.Bytes;  
import org.jsoup.Jsoup;  
import org.jsoup.nodes.\*;  
import org.jsoup.select.Elements;  
  
import Java.io.File;  
import Java.io.IOException;  
import Java.util.\*;  
  
/\*\*  
 \* author:you\_shoubian time:2016-1-19  
 \*/  
public class TaobaoSearch {  
 private static final Configuration configuration = HBaseConfiguration.create();  
 private static Connection connection;  
 static {

configuration.set("hbase.zookeeper.quorum", "slave1:2181,slave2:2181,slave3:2181");  
 configuration.set("zookeeper.znode.parent", "/hbase");  
  
 try {  
 connection = ConnectionFactory.createConnection(configuration);  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 public static void main(String args[]) {  
 try{  
 getItemsInfoList("包邮",2);  
 }catch (Exception e){  
 e.printStackTrace();  
 }  
  
 }  
  
 public static List<ItemsInfo> getItemsInfoList(String searchWord, int pNum) throws IOException {  
 List<ItemsInfo> itemsInfoList = new ArrayList<ItemsInfo>();  
 TableName tbname = TableName.valueOf("indexinfo");  
 Table table = null;  
 Result result = null;  
 table = connection.getTable(tbname);  
 result = table.get(new Get(Bytes.toBytes(searchWord)));  
 byte[] byteURLs = result.getValue(Bytes.toBytes("info"), Bytes.toBytes("val"));  
 String URLs = "";  
 if(byteURLs!=null && byteURLs.length>0)  
 {  
 URLs = new String(byteURLs);  
 }  
  
 //带个数的URL(次数##URL),每个URL中间用####隔开  
 String[] comURLarr = null;  
 if(URLs != null && URLs.length()>0)  
 {  
 comURLarr = URLs.split("####");  
 }  
  
 Map<String,Integer> URLmap = new HashMap<>();  
 for(String URLandnum : comURLarr){  
 String[] tim = URLandnum.split("##");  
 URLmap.put(tim[1],Integer.parseInt(tim[0]));  
 }  
 //查询商品信息表  
 if(comURLarr != null && comURLarr.length>0)  
 {  
 TableName tbname2 = TableName.valueOf("goodinfo");  
 Table table2 = null;  
 table2 = connection.getTable(tbname2);  
 List<Get> list = new ArrayList<Get>();  
  
 Get get = null;  
 for(String comURL : comURLarr)  
 {  
 String[] arr = comURL.split("##");  
 if(arr!=null && arr.length>1){  
 get = new Get(Bytes.toBytes(arr[1]));  
 list.add(get);  
 }  
 }  
  
 Result[] results = table2.get(list);  
  
 for(Result re : results)  
 {  
 System.out.println(Bytes.toString(re.getRow()) + ": " + Bytes.toString(re.getValue(Bytes.toBytes("info"), Bytes.toBytes("itemTitle"))));  
 }  
 List<Object[]> itemsList = new ArrayList<>();  
  
 for(Result res : results){  
 ItemsInfo item = new ItemsInfo();  
 Object[] item2 = new Object[2];  
 item.setItemAddress(Bytes.toString(res.getRow()));  
  
 item.setItemTitle(Bytes.toString(res.getValue(Bytes.toBytes("info"),Bytes.toBytes("itemTitle"))));  
 item.setItemShop(Bytes.toString(res.getValue(Bytes.toBytes("info"),Bytes.toBytes("itemShop"))));  
 item.setItemShopLocaltion(Bytes.toString(res.getValue(Bytes.toBytes("info"),Bytes.toBytes("itemShopLocaltion"))));  
 item.setItemPrice(Bytes.toString(res.getValue(Bytes.toBytes("info"),Bytes.toBytes("itemPrice"))));  
 item.setItemDealCnt(Bytes.toString(res.getValue(Bytes.toBytes("info"),Bytes.toBytes("itemDealCnt"))));  
 item2[0] = item;  
 item2[1] = URLmap.get(item.getItemAddress());  
 itemsList.add(item2);  
 }  
 Collections.sort(itemsList, new Comparator<Object[]>() {  
 @Override  
 public int compare(Object[] o1, Object[] o2) {  
 if(o1.length != 2 || o2.length != 2 ){  
 return 0;  
 }  
 if((int)o1[1] < (int)o2[1]){  
 return 1;  
 }else{  
 return -1;  
 }  
 }  
 });  
  
  
 for(Object[] objarr : itemsList.subList(0,Math.min(pNum,itemsList.size()))){  
 System.out.println(objarr[1]);  
 itemsInfoList.add((ItemsInfo) objarr[0]);  
 }  
 }  
  
 return itemsInfoList;  
 }  
  
}

创建GetTaobaoDataAction servlet类，接收/getTaobaoData的http请求。

package webDataAna;  
  
import TaobaoCrawler.ItemsInfo;  
import TaobaoCrawler.TaobaoSearch;  
  
import javax.servlet.ServletException;  
import javax.servlet.annotation.WebServlet;  
import javax.servlet.http.HttpServlet;  
import javax.servlet.http.HttpServletRequest;  
import javax.servlet.http.HttpServletResponse;  
import java.io.IOException;  
import java.util.ArrayList;  
import java.util.List;  
  
/\*\*  
 \* Servlet implementation class GetTaobaoDataAction  
 \*/  
  
@WebServlet("/getTaobaoData")  
public class GetTaobaoDataAction extends HttpServlet {  
 private static final long serialVersionUID = 1L;  
   
 /\*\*  
 \* @see HttpServlet#HttpServlet()  
 \*/  
 public GetTaobaoDataAction() {  
 super();  
 // TODO Auto-generated constructor stub  
 }  
  
 /\*\*  
 \* @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)  
 \*/  
 protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {  
 // TODO Auto-generated method stub  
 doPost(request, response);  
 }  
  
 /\*\*  
 \* @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse response)  
 \*/  
 protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {  
 // TODO Auto-generated method stub  
 response.setContentType("text/html;charset=UTF-8");  
 response.setHeader("content-type", "text/html;charset=utf-8");  
 response.setCharacterEncoding("UTF-8");  
   
 try {  
 String searchWord = new String(request.getParameter("searchWord").getBytes("iso-8859-1"), "utf-8");  
 String pageNumStr = new String(request.getParameter("pageNum").getBytes("iso-8859-1"), "utf-8");  
 int pageNum = 1;  
 try {  
 pageNum = Integer.parseInt(pageNumStr);  
   
 } catch (NumberFormatException e) {  
 // TODO: handle exception  
 System.out.println("页数不是整数，默认为1");  
 pageNum = 1;  
 }  
   
 if(searchWord.isEmpty()){  
 searchWord = "无搜索词，结果随机搜索";  
 }  
   
 //System.out.println("/"+searchWord+"/ " + pageNum+" "+searchWord.isEmpty());  
 List<ItemsInfo> itemsInfoList = null;  
 try {  
 itemsInfoList = TaobaoSearch.getItemsInfoList(searchWord, pageNum);  
 }catch (Exception e){  
 e.printStackTrace();  
 itemsInfoList = new ArrayList<>();  
 }  
 int itemsLen = itemsInfoList.size();  
 request.removeAttribute("itemsInfoList");  
 request.removeAttribute("searchWord");  
 request.removeAttribute("pageNumStr");  
 request.removeAttribute("pageNum");  
   
   
 request.setAttribute("itemsInfoList", itemsInfoList);   
 request.setAttribute("searchWord", searchWord);  
 request.setAttribute("pageNumStr", pageNumStr);  
 request.setAttribute("pageNum", pageNum);  
   
 request.getRequestDispatcher("TaobaoIndex.jsp").forward(request, response);  
   
  
 } catch (Exception e) {  
 // TODO: handle exception  
 }  
 }  
  
}

至此这个项目的全部代码已经部署完毕，你只需要将这JavaWeb项目部署到Tomcat上就可以运行了，但是在运行之前还需要配置一下你本机的Hosts文件，用于连接ZooKeeper。将你的集群中的机器的hostname和ip添加到你的本地中host文件。

38.3.3 网页扒取

因为淘宝在自己的网页上采用了反爬虫技术，用Java代码不容易扒取到正确的数据，所以我们需要采用一些外部的插件去获取淘宝网页，为了使用这些外部的插件我们需要提前准备做一些准备。

我们首先需要从服务器上使用ftp下载phantomjs-1.9.8-windows，n1k0-casperjs-1.1-beta3-0-g4f105a9，然后将他们添加到本地的PATH中，对phantomjs-1.9.8-windows，只需要将phantomjs-1.9.8-windows/目录下的phantomjs-1.9.8-windows加入到本地PATH中，n1k0-casperjs-1.1-beta3-0-g4f105a9要将n1k0-casperjs-4f105a9\bin添加到本地PATH中。

然后再从服务器上下载casperjs文件夹，把这个目录放到D盘的根目录下，然后在D盘的根目录建立一个tempHTML目录然后在里面添加temp.html文件。­

至此，准备工作就做完了。

接下来我们打开Tomcat，然后访问spider.jsp页面。如图38-3所示：

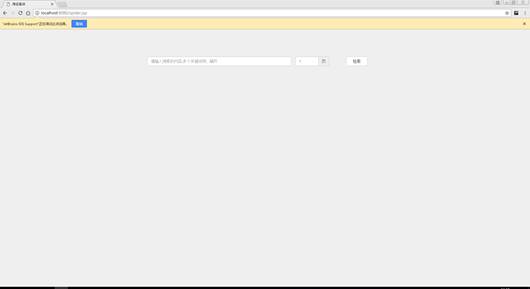


图38-3 访问spider.jsp页面

输入你想要扒取的关键词，可以同时扒取多个关键词，用逗号隔开。输入扒取的页数。稍等一会就可以看到扒取的结果（因为淘宝对同一个IP的单位时间淘宝页面的次数，所以当扒取的数量比较多的时候会比较慢。）

注：如果实验室无法连接外网，可以下载/root/data/38/data.csv文件，然后利用前面学习的HBase的导入数据的知识将数据导入到HBase中，文件以”,”分割。表名为goodinfo，和indexinfo。

38.3.4 建立关键词索引

上节扒下来的数据只是存放在HBase中goodinfo表中，这个时候没有建立索引，我们在查询的时候是从索引表中根据关键词检索出跟关键词有关的URL，并按照出现的次数按照相关性大小排序。所以我们需要将扒下来的数据建立索引，然后才能正确的使用。

在创建索引之前还需要做一些准备工作。因为创建索引的时候需要根据全文做分词提取关键词，然后根据关键词和链接的关系才能正确建立索引，所以需要分词工具的支持。分词工具支持的jdk为jdk1.8。所以我们在创建索引之前需要把我们的Hadoop所依赖的jdk版本设置为jdk1.8。修改步骤如下。

先解压 ~/data/38/jdk-8u121-linux-x64.tar.gz 文件。解压命令：

[root@client 38]# tar -zxvf jdk-8u121-linux-x64.tar.gz

该命令将jdk1.8的压缩包直接解压在38目录下。

然后修改Hadoop集群中的配置文件hadoop-env.sh（所有机器均需要修改）。修改命令如下

[root@master hadoop]# vim /usr/cstor/hadoop/etc/hadoop/hadoop-env.sh

修改内容如下：

export JAVA\_HOME=/root/data/38/jdk1.8.0\_12

然后重启Hadoop集群即可。

我们建立索引的方式式才用的MapReduce的方法。我们将createindex这个项目打包，然后向Hadoop中提交这个任务即可。Createindex会根据商品的描述的分词结果建立关键词对URL的索引。所以我们还需要在建立分词的时候调用分词的数据，分词的数据存放在/root/data/38/data。 你需要将这个文件夹拷贝到运行jar的机器的根目录下。

拷贝命令如下。

[root@client ~]# cp -r /root/data/38/data /

向Hadoop提交任务的命令如下：

[root@client hadoop]# bin/hadoop jar createindex.jar

38.3.5 关键词搜索

通过浏览器访问TaobaoIndex.jsp。界面如图38-4所示：

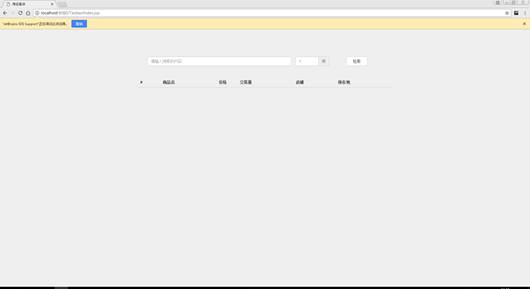


图38-4 访问TaobaoIndex.jsp页面

输入搜索的内容，并输入想选取的项数，然后点击检索。就会弹出你想了解的关键词的数据。

38.4 实验结果

（1）爬取结果类似如图38-5所示。



图38-5 爬取结果

（2）建立索引任务成功如图38-6所示。

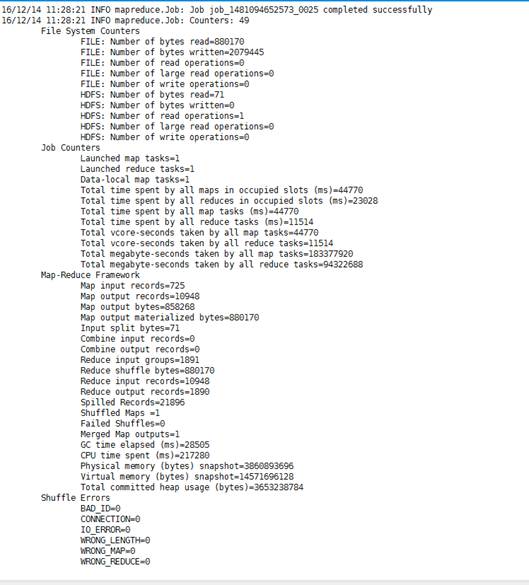


图38-6 建立索引任务

（3）搜索结果展示如图38-7所示。

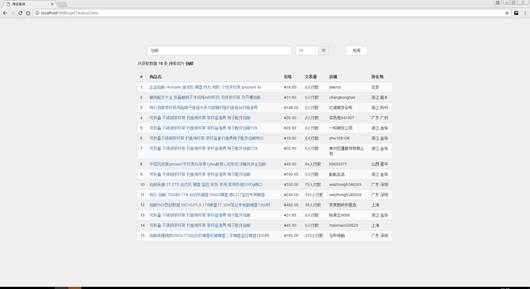


图38-7 搜索结构

实验三十九 综合实战：推荐系统

39.1 实验目的

了解常用的基于矩阵分解的协同过滤推荐算法的基本原理，掌握Spark MLlib中对基于模型的协同过滤算法的封装函数的使用，对Spark中机器学习模块内容加深理解。

39.2 实验要求

1． 实验提供数据集，包含用户数据、电影数据、电影评分数据以及我的评分数据；

2． 根据提供的电影评分数据，利用Spark进行训练，得到一个最佳推荐模型；

3． 用实际数据和平均值这两方面评价该模型的准确度；

4． 根据我的评分数据向我推荐10部电影。

39.3 实验步骤

39.3.1 试验原理概述

协同过滤算法按照数据使用，可以分为：

（1）基于用户（UserCF）；

（2）基于商品（ItemCF）；

（3）基于模型（ModelCF）。

按照模型，又可以分为：

（1）最近邻模型：基于距离的协同过滤算法；

（3）Latent Factor Mode（SVD）：基于矩阵分解的模型；

（3）Graph：图模型，社会网络图模型。

本次实验，使用的协同过滤算法是基于矩阵分解的模型，就是基于样本的用户喜好信息，训练一个推荐模型，然后根据实时的用户喜好的信息进行预测，计算推荐。

ALS是alternating least squares的缩写，意为交替最小二乘法，该方法常用于基于矩阵分解的推荐系统中。对于一个R（观众对电影的一个评价矩阵）可以分解为U（观众的特征矩阵）和V（电影的特征矩阵），在这个矩阵分解的过程中，评分缺失项得到了填充，也就是说我们可以基于这个填充的评分来给用户最适合的商品推荐了。

MLlib支持基于模型的协同过滤算法，其中user和product对应图中的user和movie，user和product之间有一些隐藏因子。MLlib使用[ALS(alternating least squares)](http://dl.acm.org/citation.cfm?id=1608614)来学习得到这些潜在因子。

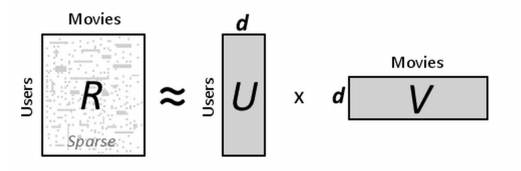


图39-1 基于矩阵分解的模型算法图

图39-1中原始矩阵R可能是非常稀疏的，但乘机UV是稠密的，即使该矩阵存在非零元素，非零元素的数量也非常少。因此模型只是对R的一种近似。原始矩阵R中大量元素是缺失的（元素值为0），算法为这些缺失元素生成（补全）了一个值，从这个角度讲，我们可以把算法称为模型。根据这个补全的矩阵，我们就可以从知道user也就知道了movies，或者知道movie也就知道了users，这就是以下实验推荐算法的基本原理。

39.3.2 数据集准备

构建模型的第一步是了解数据，对数据进行解析或转换，以便在Spark中作分析。Spark MLlib的ALS算法要求用户和产品的ID必须都是数值型，并且是32位非负整数，以下准备的数据集完全符合Spark MLlib的ALS算法要求，不必进行转换，可直接使用。

在本地目录/root/data/39/movie下有本次实验数据集，文件列表如图39-2所示。

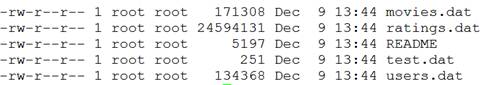


图39-2 数据集文件列表

各文件数据格式如下（详细见README文件）：

（1）用户数据（users.dat）

用户ID::性别::年龄::职业编号::邮编。

6031::F::18::0::45123

6032::M::45::7::55108

6033::M::50::13::78232

6034::M::25::14::94117

6035::F::25::1::78734

6036::F::25::15::32603

6037::F::45::1::76006

6038::F::56::1::14706

6039::F::45::0::01060

6040::M::25::6::11106

（2）电影数据（movies.dat）

电影ID::电影名称::电影种类。

3943::Bamboozled (2000)::Comedy

3944::Bootmen (2000)::Comedy|Drama

3945::Digimon: The Movie (2000)::Adventure|Animation|Children's

3946::Get Carter (2000)::Action|Drama|Thriller

3947::Get Carter (1971)::Thriller

3948::Meet the Parents (2000)::Comedy

3949::Requiem for a Dream (2000)::Drama

3950::Tigerland (2000)::Drama

3951::Two Family House (2000)::Drama

3952::Contender, The (2000)::Drama|Thriller

（3）评分数据（ratings.dat）

用户ID::电影ID::评分::时间。

6040::2022::5::956716207

6040::2028::5::956704519

6040::1080::4::957717322

6040::1089::4::956704996

6040::1090::3::956715518

6040::1091::1::956716541

6040::1094::5::956704887

6040::562::5::956704746

6040::1096::4::956715648

6040::1097::4::956715569

（4）我的评分数据（test.dat）

用户ID::电影ID::评分::时间。

0::780::4::1409495135

0::590::3::1409495135

0::1210::4::1409495135

0::648::5::1409495135

0::344::3::1409495135

0::165::4::1409495135

0::153::5::1409495135

0::597::4::1409495135

0::1580::5::1409495135

将以上数据文件上传到HDFS文件系统：

cd /usr/cstor/hadoop/bin

hdfs dfs -copyFromLocal /root/data/39/movie/ /

39.3.3 代码实现

为防止shell端INFO日志刷屏，影响查看打印信息，修改打印日志级别，进入Spark安装的conf目录下，将log4j.properties.template文件复制一份，命名log4j.properties文件，然后将文件如下配置项：

log4j.rootCategory=WARN, console

进入Spark安装目录下bin目录，启动spark-shell：

cd /usr/cstor/spark/bin

./spark-shell --master spark://master:7077

具体代码如下：

/\*\* 导入Spark机器学习推荐算法相关包 \*\*/

import org.apache.spark.mllib.recommendation.{ALS, Rating, MatrixFactorizationModel}

import org.apache.spark.rdd.RDD

/\*\* 定义函数，校验集预测数据和实际数据之间的均方根误差，后面会调用此函数 \*\*/

def computeRmse(model:MatrixFactorizationModel,data:RDD[Rating],n:Long):Double = {

val predictions:RDD[Rating] = model.predict((data.map(x => (x.user,x.product))))

val predictionsAndRatings = predictions.map{ x =>((x.user,x.product),x.rating)}

.join(data.map(x => ((x.user,x.product),x.rating))).values

math.sqrt(predictionsAndRatings.map( x => (x.\_1 - x.\_2) \* (x.\_1 - x.\_2)).reduce(\_+\_)/n)

}

/\*\* 加载数据 \*\*/

//1、我的评分数据(test.dat),转成Rating格式，即用户id，电影id，评分

val myRatingsRDD = sc.textFile("/movie/test.dat").map {

line =>

val fields = line.split("::")

// format: Rating(userId, movieId, rating)

Rating(fields(0).toInt, fields(1).toInt, fields(2).toDouble)

}

//2、样本评分数据(ratings.dat)，其中最后一列Timestamp取除10的余数作为key，Rating为值，

即(Int，Rating)，以备后续数据切分

val ratings = sc.textFile("/movie/ratings.dat").map {

line =>

val fields = line.split("::")

// format: (timestamp % 10, Rating(userId, movieId, rating))

(fields(3).toLong % 10, Rating(fields(0).toInt, fields(1).toInt, fields(2).toDouble))

}

//3、电影数据(movies.dat)(电影ID->电影标题)

val movies = sc.textFile("/movie/movies.dat").map {

line =>

val fields = line.split("::")

// format: (movieId, movieName)

(fields(0).toInt, fields(1))

}.collect().toMap

/\*\* 统计所有用户数量和电影数量以及用户对电影的评分数目 \*\*/

val numRatings = ratings.count()

val numUsers = ratings.map(\_.\_2.user).distinct().count()

val numMovies = ratings.map(\_.\_2.product).distinct().count()

println("total number of rating data: " + numRatings)

println("number of users participating in the score: " + numUsers)

println("number of participating movie data: " + numMovies)

/\*\* 将样本评分表以key值切分成3个部分，分别用于训练（60%，并加入我的评分数据）、

校验（20%）以及测试（20%） \*\*/

//定义分区数，即数据并行度

val numPartitions = 4

//因为以下数据在计算过程中要多次应用到，所以cache到内存

//训练数据集，包含我的评分数据

val training = ratings.filter(x => x.\_1 < 6).values.union(myRatingsRDD).repartition(numPartitions).persist()

//验证数据集

val validation = ratings.filter(x => x.\_1 >= 6 && x.\_1 < 8).values.repartition(numPartitions).persist()

//测试数据集

val test = ratings.filter(x => x.\_1 >= 8).values.persist()

//统计各数据集数量

val numTraining = training.count()

val numValidation = validation.count()

val numTest = test.count()

println("the number of scoring data for training) (including my score data): " + numTraining)

println("number of rating data as validation: " + numValidation)

println("number of rating data as a test: " + numTest)

/\*\* 训练不同参数下的模型，获取最佳模型 \*\*/

//设置训练参数及最佳模型初始化值

//模型的潜在因素的个数，即U和V矩阵的列数，也叫矩阵的阶

val ranks = List(8, 12)

//标准的过拟合参数

val lambdas = List(0.1, 10.0)

//矩阵分解迭代次数，次数越多花费时间越长，分解的结果也可能会更好

val numIters = List(10, 20)

var bestModel: Option[MatrixFactorizationModel] = None

var bestValidationRmse = Double.MaxValue

var bestRank = 0

var bestLambda = -1.0

var bestNumIter = -1

//根据设定的训练参数对训练数据集进行训练

for (rank <- ranks; lambda <- lambdas; numIter <- numIters) {

//计算模型

val model = ALS.train(training, rank, numIter, lambda)

//计算针对校验集的预测数据和实际数据之间的均方根误差

val validationRmse = computeRmse(model, validation, numValidation)

println("Root mean square: " + validationRmse + " Parameter: --rank = "

+ rank + " --lambda = " + lambda + " --numIter = " + numIter + ".")

//均方根误差最小的为最佳模型

if (validationRmse < bestValidationRmse) {

bestModel = Some(model)

bestValidationRmse = validationRmse

bestRank = rank

bestLambda = lambda

bestNumIter = numIter

}

}

/\*\* 用训练的最佳模型预测评分并评估模型准确度 \*\*/

//训练完成后，用最佳模型预测测试集的评分，并计算和实际评分之间的均方根误差（RMSE）

val testRmse = computeRmse(bestModel.get, test, numTest)

println("Optimal model parameters --rank = " + bestRank + " --lambda = " + bestLambda

+ " --numIter = " + bestNumIter +

" \nThe root mean square between the predicted data and the real data under the optimal model: " + testRmse + ".")

//创建一个用均值预测的评分，并与最好的模型进行比较，

这个mean（）方法在DoubleRDDFunctions中，求平均值

val meanRating = training.union(validation).map(\_.rating).mean

val baselineRmse = math.sqrt(test.map(x => (meanRating - x.rating) \* (meanRating - x.rating))

.reduce(\_ + \_) / numTest)

println("Root mean square between mean prediction data and real data: " + baselineRmse + ".")

val improvement = (baselineRmse - testRmse) / baselineRmse \* 100

println("The accuracy of the prediction data of the best model with respect to the mean prediction data: " + "%1.2f".format(improvement) + "%.")

//向我推荐十部最感兴趣的电影

val recommendations = bestModel.get.recommendProducts(0,10)

//打印推荐结果

var i = 1

println("10 films recommended to me:")

recommendations.foreach { r =>

println("%2d".format(i) + ": " + movies(r.product))

i += 1

}

39.4 实验结果

代码执行过程中打印日志信息如下：

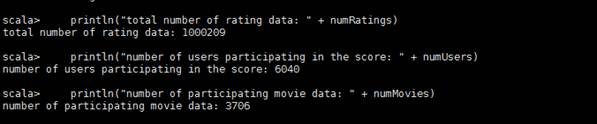


图39-3 所有数据数量统计图

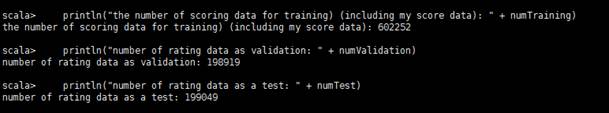


图39-4 评分数据切分的各数据集统计图

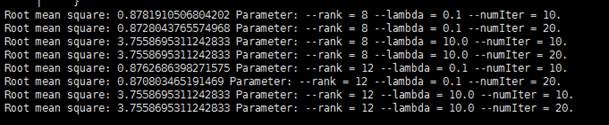


图39-5 训练时的参数及对应的误差图

说明: C:\Users\dongguangming\AppData\Roaming\Tencent\Users\1056764180\QQ\WinTemp\RichOle\OON3CTTI)LC]7Q6ZWK_VP}T.png

图39-6 最佳模型的参数及对应的误差图

https://bd.cstor.cn/experiment/exp39.files/image007.jpg

】

图39-7 均值预测的误差图

说明: C:\Users\dongguangming\AppData\Roaming\Tencent\Users\1056764180\QQ\WinTemp\RichOle\BKI]65266DIM]B4RVLD)M1E.png

图39-8 最佳模型预测相比均值预测比较图

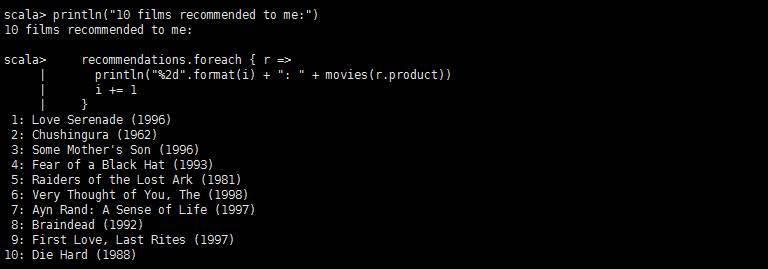


图39-9 最佳模型下向我推荐的电影图

实验四十 综合实战：环境大数据

40.1 实验目的

1． 学会分析环境数据文件；

2． 学会编写解析环境数据文件并进行统计的代码；

3． 学会进行递归MapReduce。

40.2 实验要求

要求实验结束时，每位学生均已在client服务器上运行从北京2016年1月到6月这半年间的历史天气和空气质量数据文件中分析出的环境统计结果，包含月平均气温、空气质量分布情况等。

40.3 实验原理

近年来，由于雾霾问题的持续发酵，越来越多的人开始关注城市相关的环境数据，包括空气质量数据、天气数据等等。

如果每小时记录一次城市的天气实况和空气质量实况信息，则每个城市每天都会产生24条环境数据，全国所有2500多个城市如果均如此进行记录，那每天产生的数据量将达到6万多条，每年则会产生2190万条记录，已经可以称得上环境大数据。

对于这些原始监测数据，我们可以根据时间的维度来进行统计，从而得出与该城市相关的日度及月度平均气温、空气质量优良及污染天数等等，从而为研究空气污染物扩散条件提供有力的数据支持。

本实验中选取了北京2016年1月到6月这半年间的每小时天气和空气质量数据（未取到数据的字段填充“N/A”），利用MapReduce来统计月度平均气温和半年内空气质量为优、良、轻度污染、中度污染、重度污染和严重污染的天数。

40.4 实验步骤

40.4.1 分析数据文件

在client服务器上执行下列命令，查看环境数据文件beijing.txt，路径在/root/data/40目录下。

[root@client ~]# more /root/data/40/beijing.txt

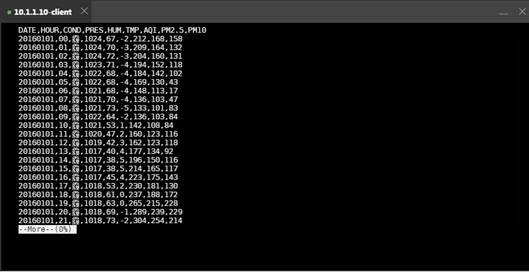


图40-1 环境数据文件格式

从图40-1中可以看到，我们需要关心的数据有第一列DATE、第二列HOUR、第六列TMP和第七列AQI。

40.4.2 将数据文件上传至HDFS

在client上传beijing.txt到HDFS的/input目录上。

[root@client ~]# hadoop fs -mkdir /input

[root@client ~]# hadoop fs -put ~/data/40/beijing.txt /input

40.4.3 编写月平均气温统计程序

在Eclipse上新建MapReduce项目，命名为TmpStat，在src目录下新建文件 TmpStat.java，并键入如下代码。

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.Reducer;

import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;

import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

import org.apache.hadoop.mapreduce.lib.partition.HashPartitioner;

public class TmpStat

{

public static class StatMapper extends Mapper<Object, Text, Text, IntWritable>

{

private IntWritable intValue = new IntWritable();

private Text dateKey = new Text();

public void map(Object key, Text value, Context context)

throws IOException, InterruptedException

{

String[] items = value.toString().split(",");

String date = items[0];

String tmp = items[5];

if(!"DATE".equals(date) && !"N/A".equals(tmp))

{//排除第一行说明以及未取到数据的行

dateKey.set(date.substring(0, 6));

intValue.set(Integer.parseInt(tmp));

context.write(dateKey, intValue);

}

}

}

public static class StatReducer extends Reducer<Text, IntWritable, Text, IntWritable>

{

private IntWritable result = new IntWritable();

public void reduce(Text key, Iterable<IntWritable> values, Context context)

throws IOException, InterruptedException

{

int tmp\_sum = 0;

int count = 0;

for(IntWritable val : values)

{

tmp\_sum += val.get();

count++;

}

int tmp\_avg = tmp\_sum/count;

result.set(tmp\_avg);

context.write(key, result);

}

}

public static void main(String args[])

throws IOException, ClassNotFoundException, InterruptedException

{

Configuration conf = new Configuration();

Job job = new Job(conf, "MonthlyAvgTmpStat");

job.setInputFormatClass(TextInputFormat.class);

TextInputFormat.setInputPaths(job, args[0]);

job.setJarByClass(TmpStat.class);

job.setMapperClass(StatMapper.class);

job.setMapOutputKeyClass(Text.class);

job.setMapOutputValueClass(IntWritable.class);

job.setPartitionerClass(HashPartitioner.class);

job.setReducerClass(StatReducer.class);

job.setNumReduceTasks(Integer.parseInt(args[2]));

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(IntWritable.class);

job.setOutputFormatClass(TextOutputFormat.class);

TextOutputFormat.setOutputPath(job, new Path(args[1]));

System.exit(job.waitForCompletion(true) ? 0 : 1);

}

}

使用Eclipse软件将TmpStat项目导出成Jar文件，指定主类为TmpStat，命名为tmpstat.jar，并上传至client服务器上。

40.4.4 查看月平均气温统计结果

在client上执行tmpstat.jar，指定输出目录为/monthlyavgtmp，reducer数量为1。如图40-2所示：

[root@client ~]# hadoop jar tmpstat.jar /input /monthlyavgtmp 1



图40-2 运行tmpstat.jar

在client上查看统计结果。如图40-3所示：

[root@client ~]# hadoop fs -ls /monthlyavgtmp

[root@client ~]# hadoop fs -cat /monthlyavgtmp/part-r-00000

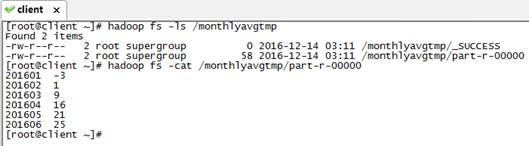


图40-3 查看月平均气温统计结果

40.4.5 编写每日空气质量统计程序

在Eclipse上新建MapReduce项目，命名为AqiStatDaily，在src目录下新建文件 AqiStatDaily.java，并键入如下代码。

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.Reducer;

import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;

import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

import org.apache.hadoop.mapreduce.lib.partition.HashPartitioner;

public class AqiStatDaily

{

public static class StatMapper extends Mapper<Object, Text, Text, IntWritable>

{

private IntWritable intValue = new IntWritable();

private Text dateKey = new Text();

public void map(Object key, Text value, Context context)

throws IOException, InterruptedException

{

String[] items = value.toString().split(",");

String date = items[0];

String aqi = items[6];

if(!"DATE".equals(date) && !"N/A".equals(aqi))

{

dateKey.set(date);

intValue.set(Integer.parseInt(aqi));

context.write(dateKey, intValue);

}

}

}

public static class StatReducer extends Reducer<Text, IntWritable, Text, IntWritable>

{

private IntWritable result = new IntWritable();

public void reduce(Text key, Iterable<IntWritable> values, Context context)

throws IOException, InterruptedException

{

int aqi\_sum = 0;

int count = 0;

for(IntWritable val : values)

{

aqi\_sum += val.get();

count++;

}

int aqi\_avg = aqi\_sum/count;

result.set(aqi\_avg);

context.write(key, result);

}

}

public static void main(String args[])

throws IOException, ClassNotFoundException, InterruptedException

{

Configuration conf = new Configuration();

Job job = new Job(conf, "AqiStatDaily");

job.setInputFormatClass(TextInputFormat.class);

TextInputFormat.setInputPaths(job, args[0]);

job.setJarByClass(AqiStatDaily.class);

job.setMapperClass(StatMapper.class);

job.setMapOutputKeyClass(Text.class);

job.setMapOutputValueClass(IntWritable.class);

job.setPartitionerClass(HashPartitioner.class);

job.setReducerClass(StatReducer.class);

job.setNumReduceTasks(Integer.parseInt(args[2]));

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(IntWritable.class);

job.setOutputFormatClass(TextOutputFormat.class);

TextOutputFormat.setOutputPath(job, new Path(args[1]));

System.exit(job.waitForCompletion(true) ? 0 : 1);

}

}

使用Eclipse软件将AqiStatDaily项目导出成Jar文件，指定主类为AqiStatDaily，命名为aqistatdaily.jar，并上传至client服务器上。

40.4.6 查看每日空气质量统计结果

在client上执行aqistatdaily.jar，指定输出目录为/aqidaily，reducer数量为3。如图40-4所示：

[root@client ~]# hadoop jar aqistatdaily.jar /input /aqidaily 3



图40-4 运行aqistatdaily.jar

在client上查看统计结果文件。如图40-5所示：

[root@client ~]# hadoop fs -ls /aqidaily



图40-5 查看aqistatdaily.jar运行结果文件

可以看到，结果文件被分成了3个部分，依次查看这3个文件的内容，即可看到每天的空气质量统计结果数据。如图40-6所示：

[root@client ~]# hadoop fs -cat /aqidaily/p\*

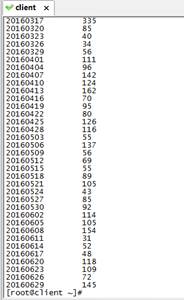


图40-6 查看每日空气质量统计结果

40.4.7 将每日空气质量统计文件进行整合

在client服务器上将每日空气质量统计结果保存到aqidaily.txt。

[root@client ~]# hadoop fs -cat /aqidaily/part-r-00000 > aqidaily.txt

[root@client ~]# hadoop fs -cat /aqidaily/part-r-00001 >> aqidaily.txt

[root@client ~]# hadoop fs -cat /aqidaily/part-r-00002 >> aqidaily.txt

[root@client ~]# cat aqidaily.txt |wc -l

182

[root@client ~]#

在HDFS上创建/aqiinput目录，并将aqidaily.txt上传至该目录下。

[root@client ~]# hadoop fs -mkdir /aqiinput

[root@client ~]# hadoop fs -put aqidaily.txt /aqiinput

40.4.8 编写各空气质量天数统计程序

在Eclipse上新建MapReduce项目，命名为AqiStat，在src目录下新建文件 AqiStat.java，并键入如下代码。

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.Reducer;

import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;

import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

import org.apache.hadoop.mapreduce.lib.partition.HashPartitioner;

public class AqiStat

{

public static final String GOOD = "优";

public static final String MODERATE = "良";

public static final String LIGHTLY\_POLLUTED = "轻度污染";

public static final String MODERATELY\_POLLUTED = "中度污染";

public static final String HEAVILY\_POLLUTED = "重度污染";

public static final String SEVERELY\_POLLUTED = "严重污染";

public static class StatMapper extends Mapper<Object, Text, Text, IntWritable>

{

private final static IntWritable one = new IntWritable(1);

private Text cond = new Text();

// map方法，根据AQI值，将对应空气质量的天数加1

public void map(Object key, Text value, Context context)

throws IOException, InterruptedException

{

String[] items = value.toString().split("\t");

int aqi = Integer.parseInt(items[1]);

if(aqi <= 50)

{

// 优

cond.set(GOOD);

}

else if(aqi <= 100)

{

// 良

cond.set(MODERATE);

}

else if(aqi <= 150)

{

// 轻度污染

cond.set(LIGHTLY\_POLLUTED);

}

else if(aqi <= 200)

{

// 中度污染

cond.set(MODERATELY\_POLLUTED);

}

else if(aqi <= 300)

{

// 重度污染

cond.set(HEAVILY\_POLLUTED);

}

else

{

// 严重污染

cond.set(SEVERELY\_POLLUTED);

}

context.write(cond, one);

}

}

// 定义reduce类，对相同的空气质量状况，把它们<K,VList>中VList值全部相加

public static class StatReducer extends Reducer<Text, IntWritable, Text, IntWritable>

{

private IntWritable result = new IntWritable();

public void reduce(Text key, Iterable<IntWritable> values,Context context)

throws IOException, InterruptedException

{

int sum = 0;

for (IntWritable val : values)

{

sum += val.get();

}

result.set(sum);

context.write(key, result);

}

}

public static void main(String args[])

throws IOException, ClassNotFoundException, InterruptedException

{

Configuration conf = new Configuration();

Job job = new Job(conf, "AqiStat");

job.setInputFormatClass(TextInputFormat.class);

TextInputFormat.setInputPaths(job, args[0]);

job.setJarByClass(AqiStat.class);

job.setMapperClass(StatMapper.class);

job.setCombinerClass(StatReducer.class);

job.setMapOutputKeyClass(Text.class);

job.setMapOutputValueClass(IntWritable.class);

job.setPartitionerClass(HashPartitioner.class);

job.setReducerClass(StatReducer.class);

job.setNumReduceTasks(Integer.parseInt(args[2]));

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(IntWritable.class);

job.setOutputFormatClass(TextOutputFormat.class);

TextOutputFormat.setOutputPath(job, new Path(args[1]));

System.exit(job.waitForCompletion(true) ? 0 : 1);

}

}

使用Eclipse软件将AqiStat项目导出成Jar文件，指定主类为AqiStat，命名为aqistat.jar，并上传至client服务器上。

40.4.9 查看各空气质量天数统计结果

在client上执行aqistat.jar，指定输出目录为/aqioutput，reducer数量为1。如图40-7所示：

[root@client ~]# hadoop jar aqistat.jar /aqiinput /aqioutput 1

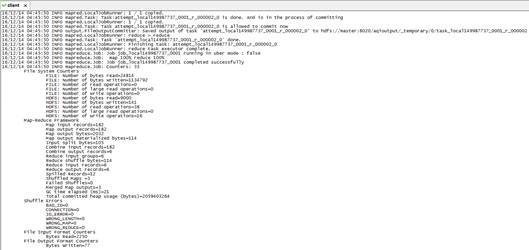


图40-7 运行aqistat.jar

在client上查看统计结果。如图40-8所示：

[root@client ~]# hadoop fs -ls /aqioutput

[root@client ~]# hadoop fs -cat /aqioutput/part-r-00000



图40-8 查看各空气质量状况天数统计结果

实验四十一 综合实战：智能硬件大数据托管

41.1 实验目的

1． 了解智能硬件的开发过程；

2． 掌握基于万物云（www.wanwuyun.com）的智能硬件流程；

3． 熟悉智能硬件数据的上传，查询，分析。

41.2 实验要求

1． 本实验最好有硬件开发板，如果条件不允许可以参考本实验的模拟器方式。为了实验通用性我们的一下步骤均采用模拟器方式为同学们展示；

2． 试验结束时同学们必须了解智能硬件的开发流程；

3． 试验结束时同学们必须掌握智能硬件的数据上传，数据查询，数据分析。

41.3 实验原理

物联网智能硬件的开发需要硬件研发和软件研发相互配合，才能实现设备的互联互通，本实验主要从软件角度去了解智能硬件的开发。

硬件设备要想实现智能化首先需要做到的是能将自己的传感器的数据传输到互联网中，我们通过收集传感器上传的数据，然后使用深度学习系统或者其他分析系统判断设备的状态，然后通过互联网平台向设备下达基于数据分析之后的最优化的命令。实际的智能硬件研发中，硬件厂商往往只能研发硬件的数据采集设备，无法研发设备数据的托管，分析平台。

万物云是物联网设备和应用的数据托管平台。智能设备可使用多种协议轻松安全地向万物云提交所产生的设备数据，在服务平台上进行存储和处理，并通过数据应用编程接口向各种物联网应用提供可靠的跨平台的数据查询和调用服务。通过使用万物云平台所提供的各项服务，用户可以收集、处理和分析互连智能设备生成的数据，在物联网应用中方便地调用这些设备数据，而无需投资，安装和管理任何基础设施，不仅大大降低了项目开发的技术门槛，缩短开发周期，而且研发和营运成本也成倍降低。如图41-1所示：



图41-1 万物云使用流程示意图

本实验主要使用java去模拟硬件设备，然后使用模拟的代码完成硬件接口调用，向万物云平台存储数据，查询分析数据。实验中将会使用一个硬件接入的接口文档和一个数据查询的客户端，这个都是可以从万物云(www.wanwuyun.com)的官方网站下载的。如图41-2所示：



图41-2 万物云功能

41.3 实验步骤

实验前先准备好实验原理中的文档和jar包，实验使用Intellij IDEA或者eclipse为IDE，先建立一个java项目，用于模拟硬件客户端。本实验主要从一下几个步骤介绍快速的智能硬件开发步骤和流程:

（1）万物云平台相关注册；

（2）建表-用于存储智能硬件的数据；

（3）智能硬件接入平台；

（4）数据上传；

（5）数据查询；

（6）简单的数据分析。

41.3.1 万物云平台相关注册

首先，登录[www.wanwuyun.com](http://www.wanwuyun.com/), 注册一个用户，如图41-3所示。



图41-3 注册界面

然后使用注册的账号登录万物云平台，然后建立智能硬件App，用于存储智能硬件设备数据，如图41-4所示。



图41-4 应用中心界面

两个重要的信息：appid：应用ID和accessid：安全管理中的安全码，在后续模拟代码中会用到。

41.3.2 建表-用于存储智能硬件的数据

点击应用，进入应用的数据表，添加智能硬件能够上报的数据列。本实验主要模拟一个带有GPS定位的温度和PM2.5测试的硬件环境设备。所以建表内容如下(建表操作可以调用SDK也可以直接在万物云平台上通过可视化界面建立，本实验采用可视化界面建立这样可以加速开发速度)，如图41-5所示。

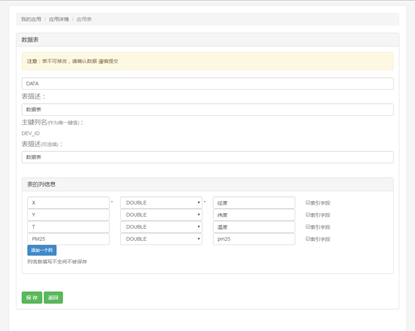


图41-5 建表

41.3.3 智能硬件接入平台

首先使用代码模拟一个硬件设备，然后完成智能设备接入平台，获取设备安全码，上报数数，查询数据，简单数据分析—以获取平均值为例。本实验代码结构如图41-6所示。

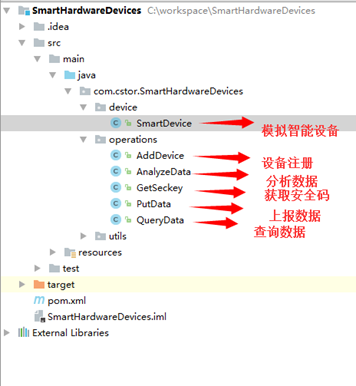


图41-6 实验代码结构

万物云提供三种协议(TCP、MQTT和Http)用于接入智能硬件，本实验使用TCP用于演示，核心代码如下：

public class AddDevice extends ChannelHandlerAdapter {

private static Logger logger = Logger.getLogger(AddDevice.class);

//发送消息给服务器

@Override

public void channelActive(ChannelHandlerContext ctx) {

logger.info("HelloClientIntHandler.channelActive");

String msg = "[{\"user\":\"smart\",\"accessid\":\"C21HCNQXNDGXNTI4MTE2MTM2\",\"aid\":\"1528613677\",\"did\":\"st002\",\"sc\":\"A\"}]";

System.out.println("设备注册信息: " + msg);

ByteBuf encoded = ctx.alloc().buffer(4 \* msg.length());

encoded.writeBytes(msg.getBytes());

ctx.writeAndFlush(encoded);

}

//接收server端的消息，并打印出来

@Override

public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {

logger.info("HelloClientIntHandler.channelRead");

ByteBuf result = (ByteBuf) msg;

byte[] result1 = new byte[result.readableBytes()];

result.readBytes(result1);

System.out.println("设备注册结果: " + new String(result1));

result.release();

ctx.close();

}

}

41.3.4 数据上传

设备注册之后，可以通过万物云平台提供的接口进行数据上报，这个是核心的服务，主要是为智能硬件提供数据存储，其核心代码如下：

//发送消息给服务器

@Override

public void channelActive(ChannelHandlerContext ctx) {

logger.info("HelloClientIntHandler.channelActive");

String msg = "[{\"seckey\":\"uwmkpo2UPHbu1RN2lgAIBw\",\"row\":{\"dev\_id\":\"st002\",\"X\":\"12.456\",\"Y\":\"13.258\",\"T\":\"15\",\"PM25\":\"25\"}}]";

System.out.println("模拟器采集的数据: " + msg);

ByteBuf encoded = ctx.alloc().buffer(4 \* msg.length());

encoded.writeBytes(msg.getBytes());

ctx.writeAndFlush(encoded);

}

//接收server端的消息，并打印出来

@Override

public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {

logger.info("HelloClientIntHandler.channelRead");

ByteBuf result = (ByteBuf) msg;

byte[] result1 = new byte[result.readableBytes()];

result.readBytes(result1);

System.out.println("数据上传结果: " + new String(result1));

result.release();

ctx.close();

}

41.3.5 数据查询

上传数据之后，可以使用万物云提供的SDK进行数据查询，希望同学们能够熟悉这个SDK的接口，能够熟练掉用其中的接口实现自己的功能，一下是查询数据的实验，起核心代码如下：

public void getData() throws TException {

//智能应用基本信息

AppInfo app = new AppInfo();

UserInfo user = new UserInfo();

user.setUserName("smart");

user.setAccessId("C21HCNQXNDGXNTI4MTE2MTM2");

app.setAppId("1528613677");

app.setUserInfo(user);

//这个是智能应用数据存放的表

String tableName = "DATA";

//组装查询条件，这里主要组装了查询的起始时间和分页大小

Conditions con = new Conditions();

PageQueryCondition page = new PageQueryCondition();

page.setPageSize(10);

Date now = new Date();

String st = Long.MAX\_VALUE - now.getTime() + "";

page.setStartPK(st);

con.setPageQueryCondition(page);

//调用万物云的客户端查询智能应用的数据

ClientDevelopApi developApi = new ClientDevelopApiService();

RePageObject ro = developApi.getTableRowsByNewPageCondition(app, "DATA", con, new ArrayList());

System.out.println("查询结果是否正确返回: " + ro.isSuccess());

System.out.println("数据结果如下: ");

for (int i = 0; i < ro.getData().size(); i++) {

Row row = (Row) ro.getData().get(i);

System.out.println(row.getValues());

}

}

41.3.6 简单的数据分析

万物云能够提供 数据托管服务，同时也能够提供数据分析的功能，本实验主要以最简单的设备数据均值分析为例，介绍如何使用万物云进行数据分析：

public void getAvgData() throws TException {

//设置智能应用的基本信息和用户的基本信息

AppInfo app = new AppInfo();

UserInfo user = new UserInfo();

user.setUserName("smart");

user.setAccessId("C21HCNQXNDGXNTI4MTE2MTM2");

app.setAppId("1528613677");

app.setUserInfo(user);

//设置开始时间

Date st = new Date();

//设备号

String devid = "st002";

//设定均值的查询

AvgQueryType avgQueryType = AvgQueryType.DAY;

//调用SDK接口

ClientDevelopApiService developApiService = new ClientDevelopApiService();

ReJSONData ros = developApiService.queryDeviceDataAvgValues(app, devid, avgQueryType, st, null,null);

System.out.println("平均值查询是否正确:"+ros.isSuccess());

System.out.println("时间:" + ros.getData().get("dateTime"));

System.out.println("平均值:" + ros.getData().get("dataArray"));

}

41.4 实验结果

本实验的每一步服务器都会返回相应的服务器代码，万物云官网是有这些代码的解释的，如上报数据的时候的返回码有下图41-7所示。

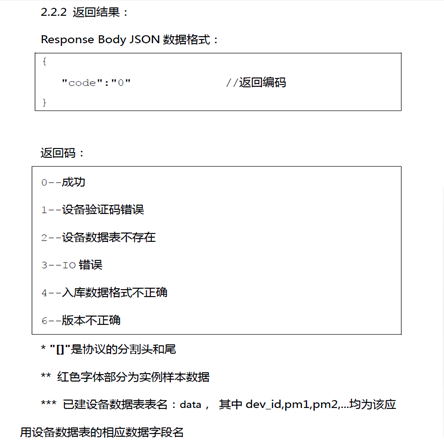


图41-7 实验结果码释义

其他每一步的代码都可以在网站上找到。

本实验做完之后是可以在万物云网站上看到知己建立的设备和设备上报的数据的，如图41-8所示。

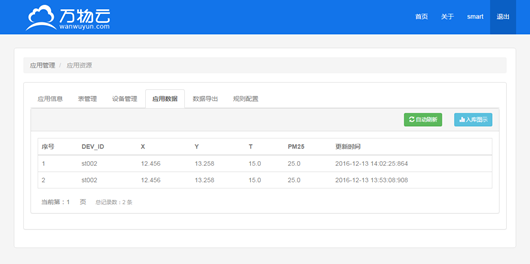


图41-8 数据查看界面

实验四十二 综合实战：贷款风险评估

42.1 实验目的

银行贷款员需要分析数据，以便搞清楚那些贷款申请者是“安全的”，银行的“风险”是什么。这就需要构造一个模型或分类器来预测类标号，其预测结果可以为贷款员放贷提供相关依据。

本次实验通过提取贷款用户相关特征（年龄、工作、收入等），使用Spark MLlib构建风险评估模型，使用相关分类算法将用户分为不同的风险等级，此分类结果可作为银行放贷的参考依据。本次实验为方便演示，选用逻辑回归算法将用户风险等级分类两类：高风险、低风险。有能力的同学可以尝试使用其他分类算法实现。

42.2 实验要求

1． 熟悉Spark程序开发流程：引入jar包—>打包开发—>打包—>提交服务器运行；

2． 熟悉Spark Mllib中分类算法的使用流程；

3． 根据教程引导，对原始数据进行分类器模型训练。

42.3 实验原理

42.3.1 分类过程及评估指标

在使用分类算法进行数据分类时，均须经过学习与分类两个阶段。

（1）学习阶段：

学习阶段按以下步骤执行：

a.选定样本数据，将该数据集划分为训练样本与测试样本两部分（划分比例自定），训练样本与测试样本不能有重叠部分，否则会严重干扰性能评估。

b.提取样本数据特征，在训练样本上执行选定的分类算法，生成分类器。

c.在测试数据上执行分类器，生成测试报告。

d.根据测试报告，将分类结果类别与真实类别相比较，计算相应的评估标准，评估分类器性能。如果性能不佳，则需要返回第二步，调整相关参数，重新执行形成新的分类器，直至性能评估达到预期要求。

（2）分类阶段

分类阶段按以下步骤执行：

a.搜集新样本，并对新样本进行特征提取。

b.使用在学习阶段生成的分类器，对样本数据进行分类。

c.判别新样本的所属类别。

42.3.2 spark-submit使用详解

将jar包提交服务器运行时，需要执行spark-submit相关命令才能运行，spark-submit 执行时命令格式如下：

|  |
| --- |
| spark-submit [options] <app jar | python file> [app options] |

需要传入的参数说明如表42-1所示。

**表42-1参数说明**

|  |  |
| --- | --- |
| 参数名称 | 具体含义 |
| --master MASTER\_URL | 可以是spark://host:port, mesos://host:port, yarn, yarn-cluster,yarn-client, local |
| --deploy-mode DEPLOY\_MODE | Driver程序运行的地方，client或者cluster |
| --class CLASS\_NAME | 主类名称，含包名 |
| --name NAME | Application名称 |
| --jars JARS | Driver依赖的第三方jar包 |
| --py-files PY\_FILES | 用逗号隔开的放置在Python应用程序PYTHONPATH上的.zip, .egg, .py文件列表 |
| --files FILES | 用逗号隔开的要放置在每个executor工作目录的文件列表 |
| --properties-file FILE | 设置应用程序属性的文件路径，默认是conf/Spark-defaults.conf |
| --driver-memory MEM | Driver程序使用内存大小 |
| --driver-library-path | Driver程序的库路径 |
| --driver-class-path | Driver程序的类路径 |
| --executor-memory MEM | executor内存大小，默认1G |
| --driver-cores NUM | Driver程序的使用CPU个数，仅限于Spark Alone模式 |
| --supervise | 失败后是否重启Driver，仅限于Spark Alone模式 |
| --total-executor-cores NUM | executor使用的总核数，仅限于Spark Alone、Spark on Mesos模式 |
| --executor-cores NUM | 每个executor使用的内核数，默认为1，仅限于Spark on Yarn模式 |
| --queue QUEUE\_NAME | 提交应用程序给哪个YARN的队列，默认是default队列，仅限于Spark on Yarn模式 |
| --num-executors NUM | 启动的executor数量，默认是2个，仅限于Spark on Yarn模式 |
| --archives ARCHIVES | 仅限于Spark on Yarn模式 |

42.4 实验相关

42.4.1 实验环境

开发工具：[IntelliJ IDEA](https://www.jetbrains.com/idea/)。

操作系统：CentOS6.5。

编程语言：Scala 2.10.4。

相关软件：Hadoop2.6.0、Spark1.6.0。

42.4.2 实验数据

数据来源于：https://www.kaggle.com/，并作部分修改，源数据见附件adult.csv。数据内容解释如下：

（1）risk-rating:0, 1；

（2）age: continuous；

（3）workclass: Private, Self-emp-not-inc, Self-emp-inc, Federal-gov, Local-gov, State-gov, Without-pay, Never-worked；

（4）fnlwgt: continuous；

（5）education: Bachelors, Some-college, 11th, HS-grad, Prof-school, Assoc-acdm, Assoc-voc, 9th, 7th-8th, 12th, Masters, 1st-4th, 10th, Doctorate, 5th-6th, Preschool；

（6）education-num: continuous；

（7）marital-status: Married-civ-spouse, Divorced, Never-married, Separated, Widowed, Married-spouse-absent, Married-AF-spouse；

（8）occupation: Tech-support, Craft-repair, Other-service, Sales, Exec-managerial, Prof-specialty, Handlers-cleaners, Machine-op-inspct, Adm-clerical, Farming-fishing, Transport-moving, Priv-house-serv, Protective-serv, Armed-Forces；

（9）relationship: Wife, Own-child, Husband, Not-in-family, Other-relative, Unmarried；

（10）race: White, Asian-Pac-Islander, Amer-Indian-Eskimo, Other, Black；

（11）sex: Female, Male；

（12）capital-gain: continuous；

（13）capital-loss: continuous；

（14）hours-per-week: continuous；

（15）native-country: United-States, Cambodia, England, Puerto-Rico, Canada, Germany, Outlying-US(Guam-USVI-etc), India, Japan, Greece, South, China, Cuba, Iran, Honduras, Philippines, Italy, Poland, Jamaica, Vietnam, Mexico, Portugal, Ireland, France, Dominican-Republic, Laos, Ecuador, Taiwan, Haiti, Columbia, Hungary, Guatemala, Nicaragua, Scotland, Thailand, Yugoslavia, El-Salvador, Trinadad&Tobago, Peru, Hong, Holand-Netherlands。

42.4.3 实验步骤

（1）IDEA配置：

在IntelliJ IDEA中需要导入Spark开发包，Spark/lib中的jar包能满足基本的开发需求，开发者可以在菜单：File->project stucture->Libraries中设置。如图42-1所示：

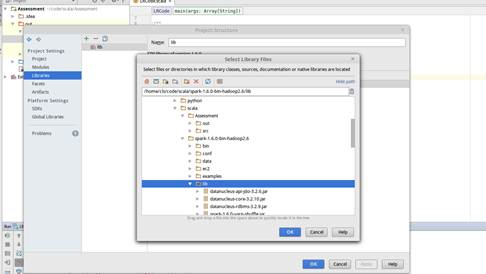


图42-1 导入Spark开发包

（2）代码步骤：

获取源数据。

|  |
| --- |
| val path = "hdfs://master:8020/input/adult.csv"  val rawData = sc.textFile(path) |

简单的数据清洗。

|  |
| --- |
| /\*\*  \* 取第一列为类标，其余列作为特征值  \*/  val data = records.map{ point =>  val firstdata = point.map(\_.replaceAll(" ",""))  val replaceData=firstdata.map(\_.replaceAll(","," "))  val temp = replaceData(0).split(" ")  val label=temp(0).toInt  val feature s = temp.slice(1,temp.size-1)  .map(\_.hashCode)  .map(x => x.toDouble)  LabeledPoint(label,Vectors.dense(features))  } |

按照一定的比例将数据随机分为训练集和测试集。

这里需要程序开发者不断的调试比例以达到预期的准确率，值得注意的是，不当的划分比例导致“欠拟合”或“过拟合”的情况产生。

|  |
| --- |
| val splits = data.randomSplit(Array(0.8,0.2),seed = 11L)  val traning = splits(0).cache()  val test = splits(1) |

训练分类模型。

|  |
| --- |
| val model = new LogisticRegressionWithLBFGS().setNumClasses(2).run(traning) |

预测测试样本的类别。

|  |
| --- |
| val predictionAndLabels = test.map{  case LabeledPoint(label,features) =>  val prediction = model.predict(features)  (prediction,label)  } |

计算并输出准确率。

|  |
| --- |
| val metrics = new BinaryClassificationMetrics(predictionAndLabels)  val auRoc = metrics.areaUnderROC()  println("Area under Roc =" + auRoc) |

输出权重最大的前10个特征。

|  |
| --- |
| val weights = (1 to model.numFeatures) zip model.weights.toArray  println("Top 5 features:")  weights.sortBy(-\_.\_2).take(5).foreach{case(k,w) =>  println("Feature " + k + " = " + w)  } |

保存与加载模型。

|  |
| --- |
| val modelPath = "hdfs://master:8020/output/"  model.save(sc, modelPath)  val sameModel = LogisticRegressionModel.load(sc,modelPath) |

（3）代码实例

|  |
| --- |
| import org.apache.spark.mllib.classification.LogisticRegressionModel  import org.apache.spark.mllib.classification.LogisticRegressionWithLBFGS  import org.apache.spark.mllib.evaluation.{BinaryClassificationMetrics, MulticlassMetrics}  import org.apache.spark.mllib.regression.LabeledPoint  import org.apache.spark.{SparkConf, SparkContext}  import org.apache.log4j.{Level, Logger}  import org.apache.spark.mllib.linalg.Vectors    object LRCode {  def main(args:Array[String]): Unit = {  val conf = new SparkConf()  .setAppName("Logisitic Test")  .setMaster("spark://master:7077")  val sc = new SparkContext(conf)    //屏蔽不必要的日志信息  Logger.getLogger("org.apache.spark").setLevel(Level.WARN)  Logger.getLogger("org.eclipse.jetty.server").setLevel(Level.OFF)    //使用MLUtils对象将hdfs中的数据读取到RDD中  val path = "hdfs://master:8020/input/adult.csv"  val rawData = sc.textFile(path)    val startTime = System.currentTimeMillis()  println("startTime:"+startTime)    //通过“\t”即按行对数据内容进行分割  val records = rawData.map(\_.split("\t"))    /\*\*  \* 取第一列为类标，其余列作为特征值  \*/  val data = records.map{ point =>  //去除集合中多余的空格  val firstdata = point.map(\_.replaceAll(" ",""))  //用空格代替集合中的逗号  val replaceData=firstdata.map(\_.replaceAll(","," "))  val temp = replaceData(0).split(" ")  val label=temp(0).toInt  val features = temp.slice(1,temp.size-1)  .map(\_.hashCode)  .map(x => x.toDouble)  LabeledPoint(label,Vectors.dense(features))    }    //按照3:2的比例将数据随机分为训练集和测试集  val splits = data.randomSplit(Array(0.8,0.2),seed = 11L)  val traning = splits(0).cache()  val test = splits(1)    //训练二元分类的logistic回归模型  val model = new LogisticRegressionWithLBFGS().setNumClasses(2).run(traning)    //预测测试样本的类别  val predictionAndLabels = test.map{  case LabeledPoint(label,features) =>  val prediction = model.predict(features)  (prediction,label)  }    //输出模型在样本上的准确率  val metrics = new BinaryClassificationMetrics(predictionAndLabels)  val auRoc = metrics.areaUnderROC()  //打印准确率  println("Area under Roc =" + auRoc)    //计算统计分类耗时  val endTime = System.currentTimeMillis()  println("endtime:"+endTime)  val timeConsuming = endTime - startTime  println("timeConsuming:"+timeConsuming)    //输出逻辑回归权重最大的前5个特征  val weights = (1 to model.numFeatures) zip model.weights.toArray  println("Top 5 features:")  weights.sortBy(-\_.\_2).take(5).foreach{case(k,w) =>  println("Feature " + k + " = " + w)  }    //保存训练好模型  val modelPath = "hdfs://master:8020/output/"  model.save(sc, modelPath)  val sameModel = LogisticRegressionModel.load(sc,modelPath)    //关闭程序  sc.stop()  }  } |

（4）服务器运行

编译器打包：

a.菜单：File->project stucture (也可以按快捷键ctrl+alt+shift+s)。

b.在弹窗最左侧选中Artifacts->左数第二个区域点击"+",选择jar，然后选择from modules with dependencies，然后会有配置窗口出现，配置完成后，勾选Build On make (make 项目的时候会自动输出jar)->保存设置。如图42-2所示：



图42-2 编译器打包

c.然后菜单：Build->make project。

d.最后在项目目录下去找输出的jar包。

代码运行:

a. hdfs中创建文件夹。

|  |
| --- |
| hadoop fs -mkdir /input  hadoop fs -mkdir /output |

b.将数据提交至hdfs。

|  |
| --- |
| hadoop fs -copyFromLocal /root/data/42/adult.csv /input |

c.将jar包提交服务器，执行以下命令。

|  |
| --- |
| ./bin/spark-submit --class LRCode --num-executors 3 --executor-memory 1g --executor-cores 3 /root/data/42/Assessment.jar |

42.5 实验结果

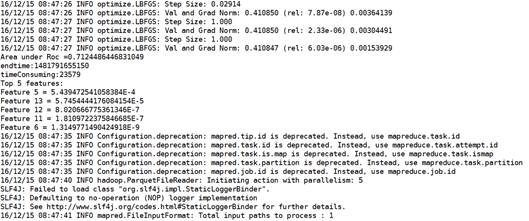


图42-3 运行结果

由图42-3可知，该分类模型准确率约为71.2%，耗时为23579毫秒，权重最大的前五个特征为第5、6、11、12、13个特征。